Improving Software Configuration Troubleshooting with Causality Analysis

by

Mona Attariyan

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2012

Doctoral Committee:

Associate Professor Jason Flinn, Chair
Professor Peter M. Chen
Assistant Professor Satish Narayanasamy
Assistant Professor Mark W. Newman

© Mona Attariyan 2012

All Rights Reserved

To my mother, father, and

two sisters, Parya and Roya,

for their unconditional love and support

ACKNOWLEDGEMENTS

First, I would like to express my sincerest gratitude to my adviser Professor Jason Flinn. I consider myself truly lucky to have worked with him during the past few years. He is an excellent researcher and mentor, and I am thankful for having such an incredible advisor.

I also owe thanks to the remaining members of my dissertation committee: Professor Peter Chen, Professor Mark Newman, and Professor Satish Narayanasamy. They all donated their time to help shape this dissertation. I would particularly like to thank Prof. Chen for his insightful comments and invaluable advice during my Ph.D. years. He is an incredible researcher, and I am grateful for the opportunity to work with him.

I was lucky to be part of an awesome research group. Ya-Yunn Su was the first member of the pervasive computing research lab that I worked with. She was an excellent friend and mentor. Dan Peek, Kaushik Veeraraghavan, and Benji Wester always made time to listen to my thoughts, questions, and complaints and very truly helpful and patient. I'd like to especially thank Mike Chow for his incredible help with the X-ray project. X-ray would never make it without his help and passion. I thank Timur Alperovich for accepting partial responsibility for breaking my two index fingers. I thank Sushant Sinha for dragging me into irrelevant and never ending discussions, which I deeply miss. I also thank Jon Oberheide, Jessica Ouyang, Brett Higgins, Jake Czyz, Kaustubh Nyalkalkar, Yunjing Xu, and Jing Zhang for having countless tea-times with me and for making 4929 the most awesome lab.

The CCCP lab, where my husband worked, was my second home at CSE. I am grateful for having incredible friends at CCCP. They were always there for me when our own lab

was too stressful to stay in. I especially thank Ganesh Dasika and Mojtaba Mehrara for wasting so much of my time waiting for my husband to finish one more round of Quake with them. I thank Mike Chu for being friendly in his own way, and Shuguang Feng, Shantanu Gupta, Mark Woh, Mehrzad Samadi, Amin Ansari, Kevin Fan, Nathan Clark, Hyoun Kyu Cho, Po-Chun Hsu, and Yongjun Park for being friendly in a more normal way.

Living in Ann Arbor would have been impossible without my great friends: Niloufar Ghafouri, Mojtaba Mehrara, Armin Alaghi, Elnaz Ansari, Mehrzad Samadi, Parisa Ghaderi, and Alireza Tabatabaeenejad. I'd especially like to thank Niloufar Ghafouri for generously letting me crash at her place for so many days. I am lucky to have such great friends and I hope that our friendships get stronger everyday.

Finally and most importantly, my family deserves major gratitude. My parents and my sisters, Parya and Roya, provided their unconditional love and encouragement through this whole process. I thank my mom for always having me in her prayers. Being away from my family was the most difficult thing during these years, and I wish I could visit them more often. Above all, I thank my husband, Amir, for his love, support, and infinite patience. I wouldn't be able to get through grad school without him, and I'm grateful to have him by my side as we discover the adventures of life.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES viii

LIST OF TABLES ix

ABSTRACT x

CHAPTER

I. Introduction 1

1.1 Causality Analysis 3
1.2 SigConf: Troubleshooting with coarse-grained causality analysis . 5
1.3 ConfAid: Troubleshooting with fine-grained causality analysis . . 6
1.4 X-ray: Troubleshooting performance problems 8
1.5 Roadmap 10

II. SigConf: Troubleshooting with Coarse-grained Causality Analysis 12

2.1 Introduction 12
2.2 Background 13
2.3 Design 15
2.4 Implementation 16
2.5 Evaluation 18
2.5.1 Methodology 19
2.5.2 Results 21
2.6 Conclusion 25

III. ConfAid: Troubleshooting with Dynamic Information Flow Analysis 26

3.1 Introduction 26
3.2 Design principles 28
3.2.1 Use white-box analysis 28
3.2.2 Operate on application binaries 30
3.2.3 Embrace imprecise analysis 30
3.3 Design and implementation 31
3.3.1 Overview: How ConfAid runs 31
3.3.2 Basic information flow analysis 33
3.3.3 Heuristics for performance 38
3.3.4 Heuristics for reducing false positives 40
3.3.5 Multi-process causality tracking 43
3.3.6 Limitations and future work 44
3.4 Evaluation 45
3.4.1 Experimental setup 45
3.4.2 Real-world misconfigurations 46
3.4.3 Effect of the weighting heuristic 51
3.4.4 Effects of bounded horizon heuristic 52
3.4.5 Random fault injection 53
3.5 Conclusion 55

IV. Deterministic Record and Replay: Taking Control of Overhead and Non-determinism 56

4.1 Design 57
4.2 Implementation 58
4.3 Evaluation 61

V. X-ray: Troubleshooting Performance Anomalies with Causality Analysis 63

5.1 Introduction 63
5.2 X-ray overview 66
5.2.1 Troubleshooting with X-ray 66
5.2.2 Mechanics of X-ray 68
5.3 Performance summarization 69
5.3.1 Basic performance summarization 69
5.3.2 Differential performance summarization 70
5.4 Implementation 70
5.4.1 Online phase 71
5.4.2 Offline phase 72
5.5 Evaluation 78
5.5.1 Experimental Setup 78
5.5.2 Root cause identification 78
5.6 Conclusion 86

VI. Related Work 87

6.1 SigConf 90
6.2 ConfAid 91
6.3 X-ray 92

VII. Conclusion 95

7.1 Future directions 95
7.1.1 Detecting anomalies 95
7.1.2 Fixing configuration problems 96
7.2 Contributions 97

BIBLIOGRAPHY 98

LIST OF FIGURES

Figure

3.1 Example to illustrate causality tracking 34

3.2 Examples illustrating ConfAid path analysis 37

3.3 Example to illustrate alternate path pruning 40

3.4 Example to illustrate the weighting heuristic 42

3.5 The effect of varying the horizon 53

4.1 Overhead of deterministic recording 61

5.1 Example of X-ray output for Apache 67

5.2 Overview of X-ray 68

5.3 Example of performance summarization 69

5.4 Example of differential performance summarization 71

5.5 An example of X-ray request extraction. The intervals marked as 1 or 2 in each process correspond to the portions of process execution that X-ray associates with the first and second requests, respectively. 73

LIST OF TABLES

Table

2.1 Description of injected bugs 20

2.2 Description of predicates for each application 21

2.3 Precision of bug diagnoses for targeted predicates 23

2.4 Precision of bug diagnoses for kitchen sink predicates 24

3.1 Description of real-world configuration bugs 47

3.2 Results for 18 real-world configuration bugs 51

3.3 Random fault injection results 54

5.1 Description of the Apache, Postfix and PostgreSQL performance test cases 79

5.2 The results for our performance test cases. 83

ABSTRACT

Improving Software Configuration Troubleshooting with Causality Analysis

by

Mona Attariyan

Chair: Jason Flinn

Software misconfigurations are time-consuming and frustrating to troubleshoot. The focus of this thesis is to reduce the time and human effort needed to troubleshoot software misconfigurations by automating the diagnosis process.

The core idea of this thesis is to automate misconfiguration diagnosis by using causality analysis to determine specific inputs to an application that cause that application to produce an undesired output. This thesis shows that we can leverage these causal relationships to determine the root cause of misconfigurations. Further, we demonstrate that it is feasible to automatically infer such relations by analyzing the execution of the application and the interactions between the application and the operating system. Based on the idea of causality analysis, we designed and developed three misconfiguration diagnosis tools: SigConf, ConfAid, and X-ray.

SigConf uses coarse-grained causality analysis to diagnose problems. The focus of SigConf is on misconfigurations that are known, i.e., the problem has been previously reported to a misconfiguration database. This database can be maintained by the developers or the users of an application. 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 in the database. SigConf deduces the state of the sick computer by running predicates that test system correctness and generating signatures based on the execution path of each predicate. SigConf generates these signatures by recording the causal dependencies of the predicate execution. For example, reading a file makes the execution causally dependent on the content of the file. SigConf compares these signatures against the signatures recorded in the database to diagnose the problem at hand.

Our second tool, ConfAid, uses a fine-grained causality analysis to diagnose misconfigurations. Compared to SigConf, ConfAid considers a narrower set of root causes, i.e. tokens in the configuration files. However, it does not require outside help to diagnose problems, and it can diagnose previously unknown misconfigurations. As the program executes, ConfAid instruments the program binaries and uses dynamic information flow analysis to extract causal dependencies introduced through data and control flow. It then uses these dependencies to link an erroneous behavior to specific configuration tokens that caused it.

While SigConf and ConfAid focus on problems that manifest as incorrect outputs, X-ray, our final tool, tackles misconfigurations that lead to performance problems. The goal of X-ray is to not only determine *what* events happened during a performance anomaly, but also infer *why* these events occurred. Similar to ConfAid, X-ray employs a fine-grained causality analysis, and it considers root causes in software configuration files, as well as input requests. X-ray introduces the technique of *performance summarization* to diagnose misconfigurations. Performance summarization first attributes performance costs to fine-grained events, such as individual instructions and system calls. It then uses dynamic information flow to determine the probable root causes for the execution of each event. The cost of each event is assigned to root causes according to the relative probability of the causes leading to the execution of that event. Finally, the total cost for each root cause is calculated by summing the per-cause costs of all events. The root cause with the highest cost is the biggest contributor to the performance anomaly.

CHAPTER I

Introduction

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.

Many studies suggest that misconfigurations are often the dominant cause of problems in deployed systems. For example, Jim Gray's classic work [32] attributes 42% of system outages to administration, while software, hardware, and environment failures account for 25%, 18%, and 14% of failures, respectively. Murphy and Gent [48] note that the percentage of failures attributable to system management is increasing over time, and that management failures have come to dominate the combination of software and hardware failures. A recent analysis of Yahoo's mission-critical Zookeeper service [37] showed that misconfigurations were accountable for the majority of all bugs exhibited. Another recent study [77] analyzed problems reported by the customers of a commercial storage company. Similar to the previous studies, configuration-related issues were the dominant cause of severe problems, causing about 31% of all failures. Other studies have shown similar results [10, 11, 50, 54]. Further, while fault tolerance techniques such as modular redundancy [45] or Byzantine fault tolerance [14] can mask some software and hardware faults, they are unlikely to help solve configuration problems caused by human error since those errors typically affect all replicas [32, 37].

Not only are configuration problems prevalent in software systems, they also have high, sometimes disastrous, impacts. For example, a recent outage in Facebook left the website inaccessible to millions of users for about 2 hours. The problem was reported to be an incorrect configuration value [36]. As another example, the entire .se domain of country Sweden was unavailable for about 1 hour, due to a DNS misconfiguration problem. The incident affected thousand of hosts and millions of users [63].

The cost of troubleshooting misconfigurations is also substantial. Technical support contributes 17% of the total cost of ownership of today's desktop computers [38], and troubleshooting misconfigurations is a large part of technical support. For information systems, administrative expenses, made up almost entirely of people costs, represent 60–80% of the total cost of ownership [23]. Even for casual computer users, troubleshooting is often enormously frustrating.

This thesis focuses on developing methods and tools that automate the troubleshooting process and thereby reduce the time to recovery (TTR) and require less manual effort by users. Misconfigurations are problems 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. Such misconfigured software might crash, produce erroneous output, or simply perform poorly.

The tools described in this thesis aim to help two types of users. End users, who may be having problems with an application on their personal computers; and system administrators, who are responsible for maintaining production systems. These users are not the developers of the application, and do not necessarily have access to the source code of the application either. Even if the source code is available, e.g. for open-source applications, inspecting the code is usually not a viable option for these users. End users may simply not have the right expertise to understand the source code. Administrators may be more familiar with the low-level code, but they usually deal with systems with various components from different vendors. Investigating the code for all these components is exceedingly diffi-

cult and time-consuming. The tools introduced in this thesis do not assume the availability of the source code. Furthermore, the outputs of these tools are high-level enough for non-developers to follow and understand. For example, our output does not contain function names or values of variables in the code.

The process of troubleshooting can be divided into two steps: diagnosing the problem, and then fixing it. Today, troubleshooting a problem is a highly manual task: First, the user collects the symptoms of the problem by inspecting the system. For instance, she may run some tests; or examine application and system logs. Expert users may be able to diagnose the problem by looking at the symptoms. If the user cannot diagnose the problem, the next step is to ask colleagues, or search online manuals, FAQ pages, and forums. The goal is to find a reported misconfiguration case that is similar or close enough to the user's case. Using a trial and error process, the user reads the often-inaccurate descriptions of problems and determines whether she is experiencing the same issue. If a solution is provided, she carefully tries the solution, hoping that the solution won't leave the system in a worst state. If the solution is wrong, it needs to be rolled back and the search continues. This process is extremely tedious and time consuming.

The goal of research in this thesis is to improve the troubleshooting process by automating it as much as possible. We specifically focus on automating diagnosis, the first task in the troubleshooting process. We have developed three different tools that diagnose various types of misconfiguration problems using different techniques. Although different, these techniques all share the common theme of causality tracking. The next section explains the idea of causality analysis and its role in improving misconfiguration troubleshooting.

1.1 Causality Analysis

Applications in today's modern systems are extraordinarily complicated. Providing sophisticated features, striving to be performant, and providing customizability and personalization are some of the factors that have added complexity to the internal logic of

applications as well as the interactions between an application and the rest of the system.

While software bugs are mostly mistakes in the internal logic of an application, misconfigurations are usually mismatches between what the user expects and what the configurable features and system state observed by the application reflects. For example, the user may expect Apache web server to serve a certain file when it receives a request; but since the file permissions are not correctly set, Apache is not able to access that file. In this example, the state of the system, i.e. the permissions of that file, mismatches the expectation of the user.

We argue that troubleshooting misconfigurations is difficult because users do not know what configuration features and system elements are read by the application as configuration inputs, and which ones are causing the application to produce an unexpected output. In the example above, if the user knows that the permissions of that specific file are read by Apache, and that input is causing Apache to deny the access, the misconfiguration is basically diagnosed.

The core idea behind this thesis is to diagnose the root cause of misconfigurations by using causality analysis to automatically infer the input elements that are causing the application to produce an incorrect output. These inputs can be tracked at different granularity. For instance, when an application reads from a file, the execution of the application becomes dependent on the content of the file. We can track this causal relationship coarsely and conclude that the execution is dependent on the entire file. A finer-grained causality analysis, on the other hand, may break the file and track smaller entities within the file, such as individual lines or words. A coarse-grained analysis is usually cheaper and faster to perform compared to a fine-grained tracking. The later, however, is more precise and can link a misconfiguration to fine-grained root causes, such as specific configuration parameters in a large configuration file.

The following statement summarizes my thesis:

Misconfiguration diagnosis is the process of determining what input elements are causing an application to produce an incorrect output. It is feasible to automatically determine such causal relations by tracking the causal dependencies between the inputs and outputs within the application execution as well as between the application and the environment. This analysis does not need the application source code, and requires no modifications to the operating system or the application.

To validate this thesis, we designed and developed three successful misconfiguration diagnosis tools. The first tool performs a coarse-grained analysis, while the other two tools use a fine-grained analysis approach. The next three sections of this chapter explain these tools in more details.

1.2 SigConf: Troubleshooting with coarse-grained causality analysis

The first part of this thesis introduces SigConf, a tool that uses coarse-grained causality analysis to diagnose a wide range of misconfiguration problems. SigConf considers known misconfigurations. These are problems that other users have already diagnosed on other machines, and have reported them to a reference computer. This computer can be maintained by the developers or the users of an application. Therefore, the problem of diagnosing an unknown bug on a *sick computer* can be reduced to identifying a state on the reference computer that is similar to the state of the sick computer.

SigConf approach to deduce this state is to run a set of predicates on the sick machine and compare the resulting execution to that generated by the same predicates on a reference computer. To effectively capture and compare the executions, SigConf generates signatures that represent the execution of a predicate by recording the causal dependencies of its execution. More specifically, SigConf records all the objects that the execution of the predicate comes to causally depend on. These objects are files, directories entries, file metadatas, and other objects read by the predicate as it runs.

SigConf causality analysis is coarse-grained and conservative. When the predicate

reads a file, SigConf considers the execution to become dependent on the entire file. SigConf does not follow the execution more closely to investigate which parts of the file are actually affecting the output. The disadvantage of a coarse-grained and conservative causality analysis approach is that it does not capture the details of the application behavior. Therefore, misconfiguration problems that follow similar execution paths may generate the same dependency set. This may adversely affect the accuracy of SigConf diagnosis. These misconfigurations can be differentiated by adding more predicates that carefully capture the difference between the two problems.

The advantage of this approach is that it creates simple and cheap signatures that are robust across different platforms. The simplicity of this approach enables us to diagnose a diverse set of misconfigurations, such as library incompatibilities, incorrect file system permissions, and wrong configuration parameters. We evaluated SigConf on three different applications: the CVS version control system, the gcc compiler suite, and the Apache Web server. We compared the diagnosis accuracy of SigConf against an algorithm that compared system states based solely on the success or failure of the predicates. SigConf significantly outperformed this algorithm, uniquely identifying the correct bug in 86–100% of the cases. Chapter II discusses design, implementation, and evaluation of SigConf in greater details.

1.3 ConfAid: Troubleshooting with fine-grained causality analysis

SigConf proved successful for misconfiguration problems that are known and are recorded in a reference computer. However, for misconfigurations that are unique to customized environments or applications for which a maintained reference machine does not exist, SigConf will not achieve much success. This issue inspired the idea of a stand-alone troubleshooting tool that does not require outside help for diagnosis.

The idea and approach of ConfAid was also influenced by our prior research project, AutoBash [66]. AutoBash troubleshoots problems by tracking causality at process and file granularity. Similar to SigConf, AutoBash treated the processes as black boxes, such that

all the outputs of a 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 may not be of great use. The lessons that we learned from SigConf and AutoBash led us to use a white-box approach for troubleshooting.

ConfAid dynamically tracks causality to identify the likely root causes of a configuration problem. When a user or administrator wants to troubleshoot a problem, such as a crash or incorrect output, she reproduces the problem while ConfAid modifies the executed application binaries to track the causal dependencies between configuration inputs and program behavior. ConfAid produces an ordered list of the configuration tokens most likely to have caused the exhibited problem. ConfAid uses dynamic information flow analysis to track causality at the level of instructions and bytes. Examining the flow of causality *within* processes as they execute, ConfAid essentially opens up the black-box of the application. Further, since ConfAid tracks causality using binary instrumentation [44], it does not require application source code to find misconfigurations.

Currently, ConfAid restricts the scope of information flow analysis to only track values that depend on data read from configuration files. ConfAid propagates dependencies by both data flow and control flow. If ConfAid determines that changing a configuration parameter may change the application's control flow such that it avoids the problem (and does not exhibit a different problem), it reports that parameter as a possible root cause.

The fine-grained and low-level analysis of ConfAid is a high-overhead activity. It imposes orders of magnitude of slow down on applications. While end user applications might be able to tolerate this slow down, it is certainly not affordable for online production software. To address this problem, we leveraged prior work in deterministic record and replay to offload the heavyweight analysis from sensitive applications. A deterministic replay

system provides a DVR-like functionality, in which an execution of a hardware or software system is recorded so that an identical execution can later be replayed on demand. Using a record and replay system, a misconfiguration can be recorded online with low overhead, while the heavy analysis happens offline on the replayed execution. Our use of deterministic replay to troubleshoot misconfigurations raised several new challenges. For instance, the fidelity of the replay must be strict enough to guarantee that the two executions are identical at the granularity observed by ConfAid. However, because the replayed execution includes analysis code that the recorded execution does not, the fidelity of the replay must be loose enough to allow the replayed execution to diverge enough to run the analysis. We show that all these goals can be achieved through careful co-design of the deterministic replay and analysis systems. Chapter IV discusses the design and implementation of our record and replay system.

We used ConfAid to troubleshoot misconfigurations in three applications: OpenSSH, the Apache Web server, and the Postfix mail server. We used two methodologies to collect the configuration problems. In the first methodology, we collected 18 real-world misconfigurations that users reported in forums and online FAQ pages. We recreated these misconfiguration cases and ran ConfAid to see if it could correctly pinpoint the root cause. ConfAid ranked the correct root cause first or second in all these cases. In the second methodology, we used ConfErr [40] to randomly generate bugs in the application's configuration file. ConfAid was able to correctly rank the root cause first or second in 55 out of 60 cases. Chapter III discusses the evaluation of ConfAid, as well as its design and implementation, in more details.

1.4 X-ray: Troubleshooting performance problems

SigConf and ConfAid tackle misconfigurations that lead to incorrect outputs. The third part of this thesis focuses on another important category of problems: misconfigurations that lead to performance anomalies. These are problems for which the outcome is cor-

rect, but the application is experiencing unusual latency or high usage of resources, such as CPU or I/O. Troubleshooting performance problems is even more challenging than troubleshooting problems with erroneous output for several reasons. First, the analysis tool must incur very low overhead, otherwise it changes the performance characteristics of the system. Further, performance problems are usually non-deterministic and transient, which make them difficult to capture and analyze.

Users and administrators typically debug performance problems by using performance monitoring tools, such as profilers and tracers. We argue that the most important reason why troubleshooting performance is challenging is that these tools only solve half of the problem. Troubleshooting a performance anomaly is essentially determining *why* certain events, such as high latency or resource usage, happened in a system. Unfortunately, most current analysis tools only determine *what* events happened during a performance anomaly — they leave the more challenging question of why those events happened unanswered. Thus, users must manually infer why the events reported by such tools happened. This step usually requires a lot of expertise and is highly tedious and time-consuming.

The goal of X-ray is to not only determine what events happened during a performance anomaly but also automatically infer why. To accomplish this, X-ray introduces the technique of *performance summarization*. Performance summarization first attributes performance costs, such as latency and I/O utilization, to fine-grained events (individual instructions and system calls). Then, it uses dynamic information flow analysis to associate each such event with a set of probable root causes such as configuration settings or specific data from input requests. The cost of each event is assigned to the potential root causes weighted by the probability that the particular root cause led to the execution of that event. Finally, the per-cause costs for all events in the program execution are summed together. The end result is a list of root causes ordered by their performance costs. This output gives the system troubleshooter a direct indication of how to fix the problem, without the need for time-consuming manual analysis. X-ray also introduces *differential performance sum-*

marization, which can be used to determine why the performance impact of two different events differed.

Similar to ConfAid, X-ray performs its analysis on the replayed execution of the application. X-ray splits its functionality among the recorded and replayed executions; for example, timestamps are captured during recording because the heavyweight analysis substantially perturbs timing. Using the deterministic record and replay system, X-ray can perform multiple rounds of analysis offline, with various scopes and metrics, on the same recorded execution.

We evaluated X-ray using three applications: the Apache Web server, the Postfix mail server, and the PostgreSQL database. We reproduced and analyzed 14 performance issues reported for these applications. In 12 of 14 cases, X-ray identified the correct root cause as the largest contributor to the performance problem; in the remaining two cases, X-ray identified the correct root cause as the third largest contributor. In chapter V, we discuss X-ray in more details.

1.5 Roadmap

The rest of this dissertation consists of the following chapters.

Chapter II describes the design, implementation , and evaluation of SigConf. SigConf uses the causal dependencies of predicate execution to detect similarities between a configuration state on a sick computer and another on a reference computer.

Chapter III describes the design, implementation, and evaluation of ConfAid. ConfAid pinpoints specific tokens in configuration files that caused an application to produce an erroneous behavior. Taking a white-box approach towards troubleshooting, ConfAid analyzes causality *within* processes as they execute. It propagates causal dependencies among multiple processes and outputs a ranked list of probable root causes.

Chapter IV discusses the design and implementation of our deterministic record and replay system. Our replay system is instrumentation-aware: it allows the analysis code to

run within the instrumented replayed execution by letting this execution diverge from the recorded execution.

Chapter V discusses the design, implementation, and evaluation of X-ray. X-ray helps users by identifying the root cause of observed performance problems. X-ray uses causality analysis to attribute the recorded performance information to root causes that include configuration options and request inputs.

Chapter VI describes related work, and chapter VII summarizes the contributions and future directions of this thesis.

CHAPTER II

SigConf: Troubleshooting with Coarse-grained Causality Analysis

2.1 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, troubleshooting a configuration problem can be both time-consuming and frustrating.

SigConf improves the troubleshooting task by automating problem diagnosis. SigConf focuses on problems that are known, i.e., the problem has been previously reported to a misconfiguration database or a reference computer. This computer can be maintained by the software developers or the application users. 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 in the reference computer and sick computers, our approach is to run a set of *predicates* that test the correctness of the computer system. In previous work [66], we used the success or failure of predicates to deduce similarity. While

this approach is intuitive, we observed 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.

SigConf proposes 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.

2.2 Background

Our previous work in configuration management, titled AutoBash [66], 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 this method 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 3.4 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.

2.3 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 [34, 80]. 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.

2.4 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.

2.5 Evaluation

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

2.5.1 Methodology

In previous work [66], 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. Table 2.1 describes the bugs that we tested. The benchmark also contains 5–8 targeted predicates for each application such that each bug causes at least one predicate to fail. These predicates are shown in Table 2.2. 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 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.

Bug	CVS configuration problem description
1	Repository not properly initialized
2	User not added to CVS group
3	CVS performs unwanted keywords substitution
4	Setgid bit not set on repository, so group for new files is incorrect
5	$TMPDIR environment variable set incorrectly
6	$CVSROOT misconfigured for a CVS user
7	$CVSROOT not set for a different CVS user
8	$CVSROOT variable set but not exported correctly
9	Repository permissions allow global access
10	Repository created using wrong group
Bug	Gcc cross-compiler problem description
1	Cross-compiler tools not in the default path
2	Cross-compiler setup overwrites default path instead of appending
3	Dangling libcrypt.so symlink does not point to correct library
4	Archive tool (ar) not in the default location
5	Kernel header module.h contains wrong content
6	Compiler cannot invoke linker due to bad location
7	Cross-compiler specs file does not contain XScale architecture definitions
8	Cross-compiler not configured to accept -pthread option
9	C compiler configured correctly, but C++ compiler is not
10	Cross-compiler not configured to pass the static link flag to the linker
Bug	Apache HTTP server problem description
1	Apache cannot search a user's home directory due to incorrect permissions
2	Apache cannot read CGI scripts due to incorrect permissions
3	Symlink used to point to CGI scripts in a user's home directory, but Apache is not configured to follow symlinks
4	Apache configuration does not allow CGI execution in user home directories
5	Misconfiguration treats CGI scripts as regular Web pages
6	Apache not configured to load PHP module
7	Handler not set for PHP pages
8	Apache not configured to use index.php as default
9	User has insufficient permission to use .htaccess authorization
10	File .htaccess in a user's home directory configured incorrectly

Table 2.1: Description of injected bugs

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:

Predicate	CVS predicate description
1	a user checks in a project and checks it out again
2	a user checks in a project, and a different user checks it out
3	same as predicate 1, but assumes a default repository is defined
4	same as predicate 3, but also checks that unauthorized users cannot access repository
5	checks if CVS performs unwanted keyword substitutions
Predicate	**gcc cross-compiler predicate description**
	Note: For all predicates, we check that the compilation succeeds and the compiled executable is the right file format
1	take a "hello world" .c file, compile it with explicit path names
2	take a "hello world" .c file, compile it using default paths
3	take a kernel module .c file, compile it
4	take a .c file, compile it, link it to a shared cryptography library
5	take several .c files, compile them into object files, archive the object files into a static library, compile a program that links in the static library
6	take a .cc file, compile it with a c++ cross compiler
7	take a .c file, compile it, statically link in a math library, check if the compilation succeeds and the compiled executable is statically linked to the math library
8	take a multi-threaded .c file, compile it for the XScale architecture
Predicate	**Apache HTTP server predicate description**
1	`wget` Apache's default home page
2	`wget` a user's default home page
3	`wget` the result of a CGI script from Apache's default root directory and `diff` the output with the expected output
4	`wget` the result of a CGI script from a user's home directory and `diff` the output with the expected output
5	`wget` the result of a PHP test page
6	`wget` a PHP test page that is set to be the default page

Table 2.2: Description of predicates for each application

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 $|R \cap C|/|C|$.

2.5.2 Results

Table 2.3 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 2.3 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 2.4 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

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 2.3: Precision of bug diagnoses for targeted 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

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 2.4: Precision of bug diagnoses for kitchen sink predicates

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.

2.6 Conclusion

SigConf, the first part of this thesis, contributes a novel method for misconfiguration diagnosis that uses the causal dependencies of predicate execution to detect similarities between a configuration state on a sick computer and another on a reference computer. We demonstrate that such information can be collected using only pre-existing system call tracing tools and without requiring application or operating system modification. Our evaluation shows that signatures generated based on these information are cheap to create and robust across different platforms.

CHAPTER III

ConfAid: Troubleshooting with Dynamic Information Flow Analysis

3.1 Introduction

The previous chapter discussed SigConf, a tool that uses causality analysis to diagnose misconfiguration problems by comparing the state of a sick machine against a reference machine. SigConf proved successful in diagnosing misconfigurations that are known and are recorded in a reference computer. However, for misconfigurations that are unique to customized environments, and for applications for which a maintained reference computer does not exist, SigConf will not achieve much success. This issue inspired the idea of a stand-alone troubleshooting tool that does not require outside help for diagnosis.

Our prior research project, AutoBash [66], also helped form the idea and approach of ConfAid. AutoBash troubleshoots problems by tracking causality at process and file granularity. Similar to SigConf, AutoBash treated the processes as black boxes, such that all the outputs of a process are considered to become 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 may not be of great use.

The lessons that we learned from SigConf and AutoBash led us to use a white-box approach for troubleshooting. In this chapter, we show that the white-box approach achieved via fine-grained information flow analysis is in fact an extremely successful approach towards troubleshooting.

This chapter introduces ConfAid, a tool that uses dynamic information flow analysis to identify the likely root causes of a configuration problem. ConfAid focuses on misconfigurations that manifest as crashes, assertion failures, or simply incorrect output. When a user or administrator wishes to troubleshoot a problem, she reproduces the problem while ConfAid modifies the executed application binaries to track the causal dependencies between configuration inputs and program behavior. ConfAid produces an ordered list of the configuration tokens most likely to have caused the exhibited problem. While dynamic analysis takes a few minutes for a complex application such as Apache, automated troubleshooting is still considerably faster and less labor-intensive than manual debugging or searching through FAQs and online forums.

ConfAid dynamically tracks causality (i.e., information flow) at a fine granularity, namely at the level of instructions and bytes. While there is a large body of work in the distributed systems community that tracks causality to understand and troubleshoot program behavior [2, 5, 6, 15, 17, 18], these prior systems essentially treat application binaries as black boxes, understanding causal relationships between processes by tracking network messages and IPCs. Some gain more information by inserting probes into applications to glean hints about their activity. ConfAid, however, "opens up the black-box" by examining the flow of causality *within* processes as they execute. Further, since ConfAid tracks causality using binary instrumentation [44], it does not require application source code to find misconfigurations.

ConfAid restricts the scope of information flow analysis to only track values that depend on data read from configuration files. ConfAid tracks dependencies introduced by both data and control flow. If it determines that altering a configuration parameter may

change the application's control flow such that it avoids the problem (and does not exhibit a different problem), it reports that parameter as a possible root cause. It propagates dependencies among multiple processes in a distributed system by annotating IPCs and network communication.

Our results show that ConfAid identifies the correct root causes of most configuration errors. We injected 18 real-world misconfigurations into OpenSSH, Apache, and the Postfix email server. ConfAid identifies the correct root cause as the most likely source of the misconfiguration in 13 cases; for the remaining 5 bugs, it lists the correct root cause as the second most likely option. ConfAid analysis takes less than 3 minutes, making the tool an attractive alternative to manual troubleshooting.

3.2 Design principles

We next briefly describe ConfAid's design principles.

3.2.1 Use white-box analysis

As mentioned before, the idea of ConfAid was partially originated from AutoBash. 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.

The granularity of the root causes reported to the user is also much finer. Instead of reporting the entire configuration file as a root cause, ConfAid points its users to specific tokens in the configuration file that it believes to be in error. This approach narrows down root causes considerably for programs like Apache.

We define a token to be a sequence of characters in the configuration file that has a specific meaning to the application. In other words, when the application recognizes this

sequence, it executes special parts of the code. For instance, the application may set a specific variable upon recognition of a certain sequence. We treat each token as a potentially discrete root cause, and we analyze its causal impacts on each byte in the process's address space as well as the process control flow.

Why did we choose a token as the smallest entity in the config file that we recognize as a discrete potential root cause? We had several choices in the granularity spectrum. One choice is the finest granularity in which each character of the config file can be a potential root cause. The problem with this choice is that individual characters usually do not have any semantic meaning to the application. For instance, the application cares about the configuration option `port`. Several lines of comments consisting of hundreds of characters that precede this option have no meaning to the application. Treating each individual character as a potential root cause produces so many unimportant causal relationships that the meaningful root causes get buried and ConfAid becomes useless.

The second option is to choose a coarser granularity. For instance, each line can be a discrete root cause. The problem with this approach is that for some applications such as Apache, a line may include multiple words where each word controls a completely different functionality and has little to do with the other words. For such configuration files, a line-based approach is clearly too coarse, because it leaves the user wondering which of the words needs to be modified.

The main problem with the above approaches is that they choose a strict granularity regardless of the application and its config file style. As mentioned earlier, the application may not care about individual characters or the line in which an option is specified in. In fact, the applications usually care about certain words or a sequence of words in the config file. For instance, the word `Protocol` has a certain meaning to OpenSSH server or Apache HTTP server treats the sequence `</Directory>` in a special way. Our approach is to identify such sequences, which we call tokens, as the application parses the config file. We call this approach a token-based approach where, based on the application, the

tokens could consist of single or multiple words. The key difference between token-based approach and previous approaches is that instead of deciding the granularity ahead of time, we choose the granularity at which the application itself parses the configuration file.

How do we identify the tokens in the config file? or how do we find out if the application cares about a sequence? We have a simple heuristic to identify sequences that the application is interested in. Applications usually compare strings read from the file against predefined constant strings in the code to find out if a certain sequence of interest exists in the config file. Whenever such comparisons happen, we conclude that the application expects the read string to have a specific meaning. At this point, we create a new token that represents that string from the file and the memory locations that contains that string become dependent on the newly created token.

3.2.2 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. ConfAid uses Pin [44] to dynamically insert instrumentation into binaries as applications run. It also uses IDA Pro [35] to statically generate control flow graphs from binaries.

3.2.3 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. Many registers and bytes in the address space would come to depend on almost all configuration values. Our prototype would identify the root cause as only one of many possible causes.

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.

3.3 Design and implementation

3.3.1 Overview: How ConfAid runs

ConfAid is designed to be used by system administrators and end users when they encounter a suspected misconfiguration that they do not know how to fix. ConfAid is run offline, once erroneous behavior has been observed. A ConfAid user reproduces the problem by executing the application while ConfAid attaches to the executing application processes and monitors information flow within them.

To use ConfAid, a user specifies: (1) which binaries ConfAid should monitor, (2) the sources of configuration data, and, as needed, (3) the erroneous external output of the application. For simple applications, ConfAid may monitor only a single process. For more complicated applications, ConfAid dynamically attaches to multiple specified processes and monitors inter-process dependencies as described in Section 3.3.5. While ConfAid could potentially monitor *any* process that receives input via IPC or a network message

from a process already monitored by ConfAid, we decided to only monitor executables specified by the user in order to limit the scope of analysis. Our prior experience with AutoBash showed that many extraneous processes communicate with processes being debugged via channels such as files, pipes, and signals, yet these processes are not needed to determine the root cause.

Similarly, we could potentially treat *any* source of input to a program as a source of configuration data. However, such an approach would dramatically slow the analysis since most locations in the process address space would come to depend on one or more inputs. In contrast, ConfAid only monitors input from designated configuration sources. This makes ConfAid analysis more tractable than generic taint tracking or program slicing because the number of locations with dependencies is small. Typically, the sources to monitor are self-evident; e.g., `httpd.conf` is the configuration source for Apache. Potentially, we could automate this process by treating all inputs from specific locations (e.g., the `etc` directory) or files with semantic keywords (such as "`*.conf`") as configuration inputs.

Finally, a ConfAid user may designate specific error conditions. ConfAid automatically treats assertion failures and exits with non-zero return codes as an erroneous terminations. However, some misconfigurations lead not to program termination, but instead to the process producing erroneous output. We therefore allow the user to specify a particular string expression as erroneous. ConfAid monitors the system calls that write to network, terminal, and other external output channels. When it finds a matching output, it considers the output an error.

ConfAid 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. By finding the needle in the haystack, ConfAid dramatically improves the total time to recovery (TTR).

3.3.2 Basic information flow analysis

In this section, we describe the basic information flow analysis used by ConfAid. For simplicity of explanation, we defer discussing optimizations and heuristics until Sections 3.3.3 and 3.3.4. We also assume that ConfAid is tracking only a single process; Section 3.3.5 describes how we extend ConfAid analysis to multiple cooperating processes on one or more computers.

ConfAid dynamically monitors the information flow from configuration sources through process memory and registers to the point in the program execution when erroneous behavior is observed. It does so by using Pin [44] to add custom logic, referred to as *instrumentation*, to the process binary. As described below, ConfAid instrumentation is executed before or after most x86 instructions executed by a monitored application.

ConfAid uses taint tracking [52] to analyze information flow. It inserts instrumentation into the application binary. 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 was specified as 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.

We use the notation, T_x to denote the taint set of variable x. T_x is a set of configuration tokens; for instance, if $T_x = \{$ `FOO, BAR` $\}$, ConfAid believes that the value of variable x could change if the user were to modify either the `FOO` or `BAR` tokens in the configuration file. In the basic information flow analysis, taints are binary (a location is either tainted by a token or it is not); in Section 3.3.4, we attach a weight to each taint.

Taint is propagated via data flow and control flow dependencies. When a monitored

```
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 3.1: Example to illustrate causality tracking

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, given the instruction $x = y + z$ where the taint sets of y and z are T_y and T_z respectively, the taint set of x, T_x, becomes $T_y \cup T_z$. 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. For example, if $T_y = \{$ `FOO, BAR` $\}$ and $T_z = \{$ `FOO, BAZ` $\}$, then $T_x = \{$ `FOO, BAR, BAZ` $\}$.

In traditional taint tracking for security purposes, control flow dependencies are often ignored to improve performance because they are harder 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. For example, on executing the statement `if (b)`, ConfAid could set the control flow taint set, T_{cf}, to $T_{cf} \cup T_b$. However, without mechanisms to *remove* taint from T_{cf}, control flow taint grows without limit. This causes too many false positives, i.e., ConfAid would identify most configuration tokens as possible root causes.

A more precise approach takes into account the basic block structure of a program. Consider the example in Figure 3.1. Assume a, b, c, and d were read from a configuration file and have taint sets T_a, T_b, T_c, and T_d, respectively (i.e., T_a is a set containing only configuration token a). 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, T_c should be removed from T_{cf} before executing $z = d$. When the program asserts, T_{cf} should only include T_d in the example, to correctly indicate that changing the value of d might fix the problem.

ConfAid also tracks implicit control flow dependencies. In Figure 3.1, 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 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. The taint set of any variable assigned on an alternate path is set to the union of its previous taint set, the taint set of the conditional, and the taint set of the variables read by the assigning instruction. In the example, $T_y = T_y \cup T_c \cup \{T_c \wedge T_b\}$. In other words, a configuration token affecting the previous value of y could change, or c could change, causing the previous value of y to be overwritten. Finally, it might be necessary for both c and b to change (as denoted by the term $\{T_c \wedge T_b\}$) since c allows the alternate assignment, and b may need to reflect a correct configuration value.

To evaluate an alternate path, ConfAid executes the program by switching the condition outcome, similar to the predicate switching approach used by Zhang et al. [88] to explore

implicit dependencies. 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. ConfAid uses Pin mechanisms to checkpoint and rollback the state of the processor, which includes the registers and CPU flags. Since some alternate paths are quite long, ConfAid uses a *bounded horizon heuristic* described in Section 3.3.3.1 to limit the number of instructions it explores along each alternate path. 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. In the example, analyzing the taken path would derive $T_x = T_a \cup T_c \cup \{T_c \wedge T_x\}$.

ConfAid also uses alternate path exploration to learn which paths avoid erroneous application behavior. ConfAid considers an alternate path 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 3.2 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),

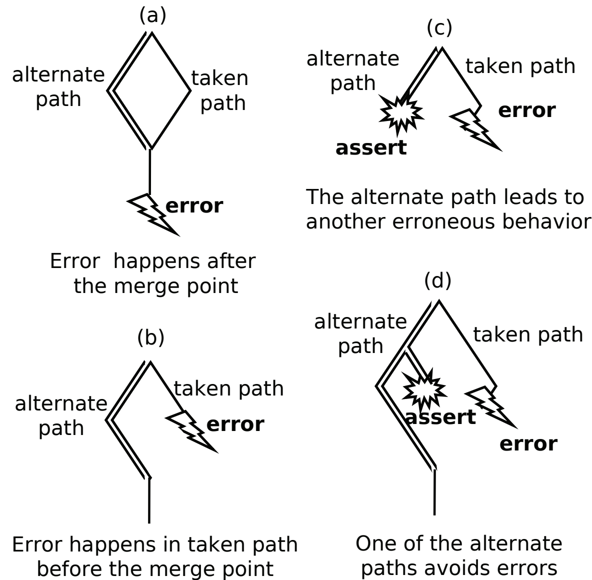


Figure 3.2: Examples illustrating ConfAid path analysis

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.

One limitation of evaluating an alternate path with predicate switching is that switching a predicate outcome, but not the underlying data values, may result in an "unnatural" execution that leads to erroneous behaviors, such as a crash due to a segmentation fault. In such circumstances, ConfAid aborts exploration of the alternate path but conservatively retains the taint of the conditional branch in the possible root causes. This conservative behavior may lead to false positives if the alternate path would in fact lead to a real error later in the execution. The early abort of the alternate path may also lead to false negatives due to unexplored variable assignments.

3.3.2.1 Abstracting library functions and system calls

There are three cases where ConfAid does not dynamically analyze information flow. The first case is when the application makes a system call. Since ConfAid does not track taint inside the operating system, the information flow analysis stops at the system call entry. The second case is commonly executed standard library functions such as `malloc` in `libc` and cryptographic functions in `libssl`. ConfAid uses a primitive static analysis for these functions to improve analysis speed while still producing the identical effect on process taint values that would have been produced by a fully-instrumented execution. Since we abstract only functions in standard libraries, such taint abstractions are application-independent. The final case is a small number of heavily optimized `libc` functions for which IDA Pro does not produce a complete static analysis.

To handle these cases, ConfAid uses *taint abstraction* of the function (or system call). A taint abstraction specifies how taint is propagated from the inputs of the functions to its outputs (e.g., return values and modified location in the address space). When a process calls one of these functions, ConfAid first executes the function without any instrumentation and then uses the taint abstraction to modify the taints of the process memory and registers.

3.3.3 Heuristics for performance

ConfAid uses two heuristics to simplify control flow analysis. These heuristics eliminate exploration of some alternate paths to concentrate on the paths that are most likely to be useful in identifying the root cause. The heuristics reduce analysis time but also introduce false positives and negatives.

3.3.3.1 The bounded horizon heuristic

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. If an alternate path contains further tainted conditional branches, ConfAid executes each path until the limit is reached. For example, if the limit is 80 instructions and a tainted conditional branch occurs after executing 50 instructions, both paths from the new branch are executed for an additional 30 instructions.

3.3.3.2 The single mistake heuristic

The second heuristic simplifies control flow analysis by assuming that the configuration file contains only a limited number of erroneous tokens. By default, ConfAid assumes that the configuration file contains a single error — we refer to this as the *single mistake* heuristic.

To illustrate how this simplifies path exploration, consider again the example in Figure 3.1. Recall that at the time the assert statement is executed, $T_x = T_a \cup T_c \cup \{T_c \wedge T_x\}$. The single mistake heuristic eliminates the last term since that term requires the values of two tokens to change simultaneously. Similarly, ConfAid derives $T_y = T_y \cup T_c$ during alternate path exploration. Note that T_y no longer depends upon T_b. This seems counterintuitive, but for the assignment $y = b$ to occur in the program, a token in T_c must change to cause the alternate path to be taken. With the single mistake heuristic, a token in T_b but not in T_c cannot be the root cause, since one token in T_c already must change.

More importantly, restricting the number of configuration values that can change reduces the alternate paths that are explored, as shown in Figure 3.3. The nested condition, $c2$, can change only if a single configuration value affects both $c1$ and $c2$. If $T_{c1} \cap T_{c2} = \emptyset$, then the alternate path of $c2$ need not be explored at all.

To implement this heuristic, we introduce a new variable, T_{alt}, that is the set of configuration options that, if changed, would cause the execution of the program to reach the

```
if (c1 == 0) { /* c1 set to 0 in config file */
    ...
} else {
    if (c2 == 0) { /* c2 set to 0 also */
        x = a;
    } else {
        y = b;
    }
}
```

Figure 3.3: Example to illustrate alternate path pruning

current instruction. Initially, T_{alt} is the set of all configuration tokens. At each condition, c, T_{alt} does not change along the taken path, but we set $T_{alt} = T_{alt} \cap T_c$ along the alternate path. In Figure 3.3, $T_{alt} = T_{c1} \cap T_{c2}$ after the second condition. When T_{alt} is $\emptyset$, the alternate path is explored no further. When a variable is assigned along an alternate path, its taint value is set to the union of its previous taint set and T_{alt}. Thus, $T_x = T_x \cup T_{c1}$ and $T_y = T_y \cup (T_{c1} \cap T_{c2})$.

The single mistake heuristic may lead to false negatives. In Figure 3.3, if $c1$ and $c2$ are tainted by a disjoint set of tokens, ConfAid will not explore the path on which y is assigned to b, so it may miss the root cause if the program later asserts based on the value of y. 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.

3.3.4 Heuristics for reducing false positives

We originally designed ConfAid to use only the basic taint tracking algorithm described in Section 3.3.2 with the bounded horizon and single mistake heuristics. However, our initial experiments with this design met with only limited success. Typically, ConfAid would include the root cause of a misconfiguration in its output set, yet the cardinality of the output set would be very large. For many bugs, ConfAid would return a significant fraction of the tokens in the configuration file.

In analyzing our initial results, we realized that it was insufficient to track information flow dependencies as binary values. 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 the implicit assumption in our design that all information flow relationships should be treated equally. Essentially, our design had no shades of gray: it either considered a location to be tainted by a token or it did not. Based on this observation, we decided to modify our design to instead track taint as a weight ranging in value between zero and one. For example, the taint of x might be represented as $\{$ FOO:w_{foo}, BAR:w_{bar} $\}$. As before, this set indicates that modifying either token FOO or BAR might change the value of x. However, if $w_{foo} > w_{bar}$, FOO has a more direct relationship to x, and hence is believed to be a better candidate for the root cause of an error that depends on x.

We revised ConfAid to use heuristics to weight the dependencies introduced by information flow differently, with those relationships that are more likely to lead to the root cause given a higher weight than those less likely to lead to the root cause. We also modified ConfAid to order the set of tokens on which an erroneous behavior depends by their respective weights before outputting them.

Our weights are based on two heuristics. First, data flow dependencies are assumed to

```
x = a;
if (c1 == 0) { /* c1 set to 0 in config file */
    y = a;
} else {
    z = b;
}
if (c2 == 0) { /* c2 set to 0 in config file */
  if (c3 == 0) { /* c3 also set to 0 */
     w = a;
  }
}
```

Figure 3.4: Example to illustrate the weighting heuristic

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.

For example, in Figure 3.4, the assignment $x = a$ is a data flow dependency, so $T_x = T_a$ (any dependencies from a are inherited at full weight). However, y inherits taint from $c1$ through a control flow dependency. Thus, $T_y = max(T_a, \frac{T_{c1}}{2})$. That is, we weight any taint from $c1$ by half, while taint inherited from a is given full weight. We use a special max operator here rather than a simple union operator, since the values are now floating point rather than binary. Specifically, $max(T_x, T_y)$ produces a set that contains all tokens that occur in either T_x and T_y. If a token appears in only one of T_x or T_y, its weight is set to its weight in that set. If a token appears in both T_x and T_y, its weight is set to the maximum of its weight in either set.

Similarly, $T_z = max(T_z, \frac{T_{c1}}{2})$ (recall that with binary values, $T_z = T_z \cup T_{c1}$ due to the single mistake heuristic). When ConfAid explores an alternate path, it replaces the

intersection operator with a corresponding min operator. Thus, in the prior example from Figure 3.3, $T_y = max(T_y, min(\frac{T_{c1}}{4}, \frac{T_{c2}}{2}))$.

Figure 3.4 also shows two nested conditions. In calculating the taint of w, condition $c3$ is considered more influential than condition $c2$ because it occurs later in the program execution. Therefore $T_w = max(T_a, \frac{T_{c3}}{2}, \frac{T_{c2}}{4})$. The same weighting applies to alternate path execution; assignments on an alternate path starting at the $c3$ branch are given twice the weight as those on an alternate path starting at the $c2$ branch.

ConfAid also weights alternate paths that avoid the erroneous behavior by their proximity to the point in application execution where the behavior is exhibited. Paths starting from the closest tainted conditional branch that avoids the erroneous behavior are given full weight, those from the next closest branch are given half weight, and so on. Note that if a configuration token has a much stronger weight on the condition of a distant branch than any tokens for closer branches, ConfAid may still rank it as the most likely root cause.

Of course, when programs do not behave as expected, ConfAid's heuristics may lead to incorrect results. For example, an application could potentially execute a substantial amount of code between the point where the erroneous behavior occurs and the point where the program outputs some value that exhibits the error (e.g., an error message). If that code contains a condition tainted by a configuration token other than the one that caused the error *and* that condition changes the specific error message that is generated, ConfAid might identify the wrong token as the most likely root cause. While such a scenario is uncommon, we did observe a single Apache bug (described further in the evaluation) in which ConfAid's heuristic failed in this manner.

3.3.5 Multi-process causality tracking

The most difficult configuration errors to troubleshoot involve multiple interacting processes. Such processes may be on a single computer, or they may reside on multiple computers connected by a network. To troubleshoot such cases, ConfAid instruments multiple

processes at the same time and propagates taint information along with the data sent when the processes communicate.

ConfAid supports processes that communicate using sockets and files. The socket support includes Unix sockets and pipes, as well as UDP and TCP sockets. ConfAid instruments the system calls that create sockets and pipes. It marks these objects as taint propagating channels if the destination is another instrumented process. Then, ConfAid intercepts all sends and receives using those channels. When data is sent, ConfAid appends a header that indicates whether or not the data is tainted and, when applicable, the exact taint of the data. Taint information is propagated at per-byte granularity if the taints of different bytes of the buffer are different. On the receiving side, ConfAid extracts the header from the received data and assigns the indicated taints to the received data.

For files, ConfAid creates an auxiliary file with a special ".confaid" extension when an instrumented process writes tainted data to a file. The auxiliary file records which bytes in the corresponding file are tainted and the specific values of those taints. Like sockets, file taint is recorded at granularities as small as one byte. For instance, the file "foo.confaid" records the tainted bytes in file "foo". When an instrumented process reads data from a file and a corresponding auxiliary file exists, ConfAid sets the taints of bytes read from the file to the values specified in the auxiliary file.

Since these operations are performed by PIN instrumentation immediately before and after system call execution, the taint propagation is hidden from the application. No operating system modifications are needed.

3.3.6 Limitations and future work

Since configuration troubleshooting is complex, ConfAid makes a number of assumptions to simplify its analysis. First, ConfAid only troubleshoots configuration problems that originate from configuration files. This limitation is not fundamental. ConfAid can be extended to track other root causes such as file system permissions and environment

variables.

Second, like previous configuration troubleshooting systems [71, 72], 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, as described in Section 3.3.3.2. However, if a file contains two incorrect tokens that represent independent mistakes, ConfAid can tackle the two errors sequentially by first identifying the token that leads to the most immediate failure, and then identifying the other token once the first error is corrected. The single mistake heuristic improves ConfAid's performance by reducing the set of possible taints tracked during dynamic analysis. In the future, we plan to allow ConfAid to track sets of two or more misconfigured tokens and measure the resulting performance overhead. Potentially, we may 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.

3.4 Evaluation

Our evaluation answers the following questions:

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

3.4.1 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 4 GB of memory. The machine runs Linux kernel version 2.6.21.

For Apache, ConfAid instruments a single process; for OpenSSH and Postfix, multiple processes are instrumented.

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 [40] to inject random errors into the configuration files of the three applications.

3.4.2 Real-world misconfigurations

We searched forums, FAQ pages and configuration documents to find actual configuration problems that users have experienced with our target applications. In total, we chose 18 misconfigurations (5–7 for each application) that were caused by errors in the configuration files. The 18 misconfigured values cover a range of data types, such as binary options, enumerated types, numerical ranges, and text entries such as server names. Table 3.1 lists these configuration errors for each application. The following section describes these errors in more details.

3.4.2.1 Description of configuration bugs

OpenSSH server:

In the first misconfiguration, the `PermitRootLogin` option is disabled in the OpenSSH server configuration file. Therefore, when users try to login as root, the server denies access, although the root password is entered correctly.

App	Bug	Description of misconfiguration
OpenSSH Server	1	The PermitRootLogin option is disabled. Therefore, the user cannot ssh as root. The server keeps denying permission although the password is entered correctly.
	2	The server only has the PasswordAuthentication option enabled, while the user can only authenticate via RSA keys.
	3	The user does not have his public key in the directory specified in the SSH server config file. Therefore, he cannot authenticate.
	4	The user is not in the AllowUsers list in the SSH config file. Therefore, he cannot connect to the server although he enters the password correctly.
	5	The MaxAuthTries option in SSH server config is set too low. Therefore, the user is disconnected if she enters her password incorrectly once.
	6	The MaxStartups options is set to 1. Therefore, the server refuses to start a new session, while another unauthenticated session is still in progress.
	7	The location of the server RSA key is not set correctly in the config file. Therefore, the client fails to verify the host key.
Apache HTTP Server	1	The path specified in the DocumentRoot option does not have a corresponding <Directory> section. Therefore, all accesses to this path are denied according to the default policy.
	2	The cgi-bin directory is ScriptAliased in the config file. This prevents the DirectoryIndex from working as expected. Therefore, the user cannot access the index file in the directory.
	3	The cgi-bin directory is aliased in the config file. However, the corresponding Directory section does not provide sufficient permissions. Therefore, accesses to this directory are denied.
	4	A virtual host with the same interface coverage is set for the HTTP server. This host points to a different DocumentRoot which overwrites the default one. Therefore, the user gets an index file with incorrect content upon accessing the server DocumentRoot.
	5	The cgi-bin directory is aliased and a CGI Handler is activated in the config file. However, the corresponding <Directory> section does not have the ExecCGI option set. The user cannot access the executables in this directory.
	6	A specific directory in DocumentRoot is also aliased to another directory outside DocumentRoot. Therefore, accesses to files in the first directory are redirected to the aliased directory, and the files are not found.
Postfix	1	The mydestination option is not set correctly in the Postfix config file. Therefore, Postfix cannot deliver mail locally.
	2	The myorigin option is set incorrectly in the Postfix config file. Therefore, the next relay host bounces the mail sent from the user's machine to the Internet.
	3	The relayhost option is set incorrectly. Therefore, Postfix cannot forward the email sent from the user's machine to the Internet.
	4	The type of alias_maps option is not supported in the user's machine. Therefore, Postfix fails to send any mail locally or to the Internet.
	5	The email address provided in luser_relay is not reachable. Therefore, Postfix cannot redirect other mail with wrong recipient to the luser_relay.

Table 3.1: Description of real-world configuration bugs

In the second bug, the OpenSSH server is configured to only allow password authentication, while the client is configured to authenticate via RSA keys. In this bug, when the

user tries to login, the password prompt does not appear, and the user gets a *permission denied* message.

In the third bug, the user tries to authenticate via RSA keys, but he does not have his public key in the directory specified in the OpenSSH server configuration file. Therefore, when the user tries to login, he receives a *permission denied* message.

In the fourth bug, the user is not listed in the `AllowUsers` list in the OpenSSH server configuration file. Therefore, when he tries to login, the server denies access, even when the user enters his password correctly.

The `MaxAuthTries` option in the OpenSSH server configuration file is set to a low number (1 in this bug) in the fifth bug. This option controls the number of incorrect authentication trials. If the user enters her password incorrectly once, she gets disconnected from the server.

For test case 6, the `MaxStartups` option is set to 1. This option controls the number of concurrent unauthenticated sessions, mainly for security purposes. In this bug, the server refuses to start a new session, because another unauthenticated session is still in progress.

In the last test case, the user cannot verify the host key of the server, because the location of the server RSA key, specified by the `HostKey` option, is incorrect in the server's configuration file. This option enables users to verify the identity of the host server, and prevents security attacks such as man-in-the-middle attack.

Apache Web server:

The Apache Web server allows users to specify configuration options for each directory, using a `<Directory>` header. In the first bug, the `DocumentRoot` option, which specifies the default path of documents in Apache, does not have a corresponding `<Directory>` section. Therefore, Apache uses the default `<Directory>` section, which denies accesses by default.

In the second bug, the cgi-bin directory is `ScriptAliased` in the Apache configuration file. This setting implies that everything in this directory is executable. Therefore, the

DirectoryIndex option in the corresponding <Directory> section will not work as expected. Thus, an attempt to access the index file, by only specifying the name of the directory, leads to an error.

In test case 3, the cgi-bin directory is aliased in the Apache configuration file. However, the access permissions are not set correctly in the corresponding <Directory> section. Therefore, all the accesses to this directory are denied.

A single Apache server can expose multiple interfaces (with different IP addresses and port numbers), using the <VirtualHost> directive. In the fourth bug , a virtual host with the same interface coverage as the default coverage is set for HTTP server. However, this host points to a different DocumentRoot, which overwrites the default one. Thus, the files that are served in the DocumentRoot path are not the files that the user expects.

In the fifth bug, the cgi-bin directory is correctly aliased in the Apache configuration file. But, the corresponding <Directory> section does not have the ExecCGI option, which instructs Apache to executes the cgi files. Thus, users cannot access executables in this directory.

In the final bug, a specific subdirectory of DocumentRoot option is aliased to another directory in the Apache configuration file. Accesses to that subdirectory are therefore redirected to the specified alias. Thus, Apache cannot find the requested files, although the user sees that they exist in the original subdirectory.

Postfix mail server:

In the first test case, Postfix cannot deliver email locally, because the mydestination option that specifies the domain name of the local machine is not set correctly.

In the second Postfix test case, the next relay host bounces the email, complaining that the domain of the sender does not exist. The reason is that the myorigin option is not set correctly in the Postfix configuration file. This option specifies the domain that locally-posted mail appears to come from.

For the third bug, the relayhost option is set incorrectly in the Postfix configuration

file. Therefore, the local SMTP process cannot forward the email from the sender to the next hop.

The alias_maps option in the Postfix configuration file specifies the alias directory that is used for local email delivery. In the fourth bug, the type of alias_maps option is not supported in the user's machine. Therefore, Postfix cannot deliver emails.

For an email with incorrect recipient addresses, Postfix tries to redirect the email to the address specified in the luser_relay option. In the fifth bug, the luser_relay address is set incorrectly. Therefore, Postfix cannot redirect the email successfully.

3.4.2.2 Results

ConfAid tracks dependencies among multiple processes for all OpenSSH and Postfix bugs. For OpenSSH, it instruments two processes that communicate via Unix sockets. For Postfix, it instruments between four and six processes that communicate via Unix sockets and files; the number of instrumented processes depend on how many processes are started before a particular bug manifests. Multi-process causality tracking is necessary to diagnose 4 out of 5 Postfix and 3 out of 7 OpenSSH bugs. For Apache, ConfAid does not track dependencies across processes since Apache starts only one process.

As shown in Table 3.2, ConfAid is highly effective in pinpointing the root cause of 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 debug 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.

The performance of ConfAid is reasonable. The time to manifest the buggy behavior varies among applications. Postfix and OpenSSH take less than 2 minutes, while Apache takes 2–3 minutes to complete. The average execution time of 1:32 minutes is much faster

Application	Bug	Total # of options	ConfAid rank of the root cause	Execution time	# false positives w/o weights
OpenSSH Server	1	47	2^{nd}(tied w/1)	1m 16s	6
	2	47	1^{st}(tied w/1)	1m 10s	1
	3	48	2nd	51s	43
	4	49	2nd	48s	44
	5	47	1st	1m 13s	43
	6	47	1st	9s	0
	7	47	1^{st}(tied w/1)	36s	43
Apache HTTP Server	1	88	2^{nd}(tied w/1)	2m 46s	87
	2	89	1st	2m 45s	87
	3	89	2^{nd}(tied w/1)	2m 45s	88
	4	93	1st	2m 59s	91
	5	89	1st	2m 46s	88
	6	89	1^{st}(tied w/1)	2m 47s	86
Postfix	1	27	1st	37s	4
	2	27	1st	1m 10s	4
	3	29	1st	47s	4
	4	29	1st	32s	2
	5	29	1st	1m 38s	0

Table 3.2: Results for 18 real-world configuration bugs

and less frustrating than trying to fix such configuration errors by looking at the logs, searching the Internet, and asking colleagues for potential clues. For instance, the 6th Apache misconfiguration in Table 3.1 is taken from a thread in linuxforums.org [42]. 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. In contrast, ConfAid identified the root cause in less than 3 minutes.

3.4.3 Effect of the weighting heuristic

We next examine the effect of the weighting heuristic introduced in Section 3.3.4. For each of the 18 real-world misconfigurations, we disabled the heuristic and re-ran ConfAid. With the heuristic disabled, ConfAid treats all sources of information flow equally. Therefore, instead of producing a ranked list of possible root causes, ConfAid returns a single set of tokens, each of which is considered equally likely to be the root cause.

The last column of Table 3.2 shows the number of false positives when the heuristic

is disabled. In every case, ConfAid identifies the correct root cause as one of the returned tokens. However, the number of other tokens returned varies substantially. Without the heuristic, there were only two misconfigurations (the 6th OpenSSH bug and the 5th Postfix bug) for which ConfAid produces no false positives. For six other bugs, the number of false positives is relatively low (less than 6). For the remaining 10 bugs, ConfAid returns almost all options as possible root causes. Thus, without the weighting heuristic, ConfAid is ineffective for 55% of the misconfigurations.

3.4.4 Effects of bounded horizon heuristic

We next investigated the effect of varying ConfAid's limit on the number of instructions executed along each alternate path (discussed in Section 3.3.3.1) from the default value of 80 instructions. As Figure 3.5 shows, varying the limit has substantially different effects on execution time, depending on the application being instrumented. For OpenSSH (bug #1), the execution time increases approximately linearly from 56 seconds with no alternate path exploration to 2:29 minutes with a horizon of 1600 instructions. On the other hand, Postfix (bug #1), shows an apparently exponential growth as the bound increases. The execution time starts at 21 seconds with no alternate path exploration and increases to 7:10 minutes for a horizon of 800 instructions. With a horizon of 1600, ConfAid analysis did not complete.

This difference in behavior derives from the nature of the applications. We found that even with a limit of 80 instructions, more than 80% of the tainted conditional branches in the OpenSSH bug reach their merge points for all alternate paths. Increasing the horizon only affects a small fraction of the branches since the rest are short enough to finish within the limit. On the other hand, for Postfix, less than 50% of the branches reach their merge point within the limit of 80 instructions. As we raise the limit, the percentage of the completed branches increases slowly to 60%.

To summarize, we found that there is no single limit that works best for all applications.

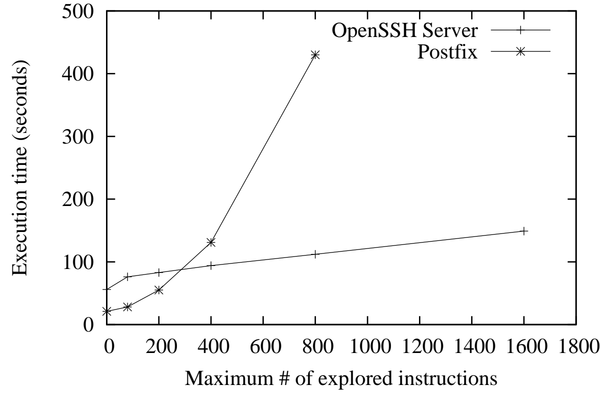


Figure 3.5: The effect of varying the horizon

Consequently, we envision that we could augment ConfAid to use an iterative search process in which it would start with a small horizon to generate results quickly, then continue to execute with larger horizons to refine the results.

3.4.5 Random fault injection

We next used ConfErr [40] to randomly generate configuration errors. 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. We then injected the errors one by one and measured the effectiveness and performance of ConfAid.

As shown in Table 3.3, ConfAid performs very well on these errors. The average time to execute all three applications is lower than the average execution time for the real-world errors used in the previous section. The main reason for this difference is that the real-world errors are often more complex than the randomly-generated ones. Therefore, it takes more time for the application to manifest the buggy behavior for real-world errors.

For the randomly generated errors, ConfAid instruments up to two processes for OpenSSH and up to six processes for Postfix. However, many faults are exhibited before these applications start additional processes; in such cases, ConfAid only instruments one process.

For OpenSSH, ConfAid successfully pinpointed the root cause (where we define suc-

App.	root causes ranked first	root causes ranked first with one tie	root causes ranked second	root causes ranked second with one tie	root causes ranked worse than second	Average 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 3.3: Random fault injection results

cess as listing the actual root cause as one of the top two options) 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 abstracting more calls.

ConfAid also successfully diagnoses 95% of the Apache errors. For the remaining error, ConfAid ranks the root cause 9th. The configuration error is that the DirectoryIndex file for the main document root is listed incorrectly in the Apache configuration file. The DirectoryIndex file is the file that Apache serves if that directory is accessed without mentioning a specific file. For instance, accessing `http://server.com/images/` will return the DirectoryIndex file listed for the `images` directory. However, the `Indexes` option is also activated for the document root directory. This option allows Apache to send the list of the files in the directory if no specific file in that directory is requested. The combination of these two options causes Apache to serve the list of files in the main document directory instead of the index file. ConfAid determines that the content sent to the user is dependent on the `Indexes` and related options first and the DirectoryIndex option next. Thus, the root cause gets ranked lower in the list. This ordering is a direct result of the heuristic discussed in Section 3.3.4 that considers branches closer to the erroneous behavior to be more likely to lead to the root cause than those farther away.

For Postfix, ConfAid diagnoses 85% of the errors effectively. The remaining 3 errors are 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. Based on these results, we plan to extend our alternate path analysis to look for tokens that could be read from the config file along branches that are not actually executed. We can taint variables modified along those branches with a value that is dependent upon the branch conditions that led to that path.

Overall, ConfAid successfully diagnosed 55 out of 60 random errors by ranking the actual root cause first or second. Out of the remaining 5 errors, we believe that 4 (the OpenSSH server error and the three Postfix errors) can be diagnosed with further improvements to the ConfAid implementation. The remaining error (the Apache error) is a direct result of our weighting heuristic and seems hard for ConfAid to diagnose correctly.

3.5 Conclusion

Misconfigurations 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.

CHAPTER IV

Deterministic Record and Replay: Taking Control of Overhead and Non-determinism

The dynamic information flow analysis that ConfAid performs is a high-overhead activity. It imposes several orders of magnitude of slow down on applications. While this slowdown may be tolerable for some desktop applications, it is certainly not affordable for production environment. Another problem is that some misconfigurations are time-sensitive, i.e. they manifest differently if the timing of the execution changes. It is difficult to correctly capture such problems with an analysis that highly perturbs the execution timings. Furthermore, some misconfigurations, especially performance-related issues discussed in the next chapter, are rare and difficult to reproduce. Therefore, the users may not be able to easily recreate them for analysis.

To address these problems, we decided to augment ConfAid with a deterministic record and replay system that offloads time-consuming analysis from the online, time-sensitive execution. A deterministic record and replay system recreates an execution by recording the initial state of the execution and logging all non-deterministic events that occur during the execution [9, 27, 64, 68]. The replay system subsequently reproduces the execution on demand by restarting execution from the initial state and supplying the previously-recorded values for all non-deterministic events. In this chapter, we discuss the differences between our system and the existing deterministic record and replay systems.

4.1 Design

While deterministic replay is a well-studied technique, we encountered several new challenges in adapting the technique to work with ConfAid. In particular, we found that we needed to carefully balance the *fidelity* of the record and replay, and that we needed to *co-design* the deterministic replay system to work with the dynamic instrumentation and analysis employed by ConfAid.

We define the fidelity of the replay to be the degree to which the replayed execution is guaranteed to match the recorded execution. For the purposes of ConfAid, replay fidelity must be high enough to guarantee that the recording and replaying systems execute the same application instructions and system calls in the same order. Since ConfAid extracts causal dependencies from the data flow and control flow of the execution, if the two executions were allowed to differ, ConfAid could provide incorrect root cause diagnosis.

On the other hand, the fidelity of replay must be low enough so that ConfAid can execute *both* application and dynamic instrumentation instructions and system calls during replay. From the point of view of the replay system, the replayed execution will contain a large number of additional events that were not present during recording.

Thus, the design of our record and replay system walks a fine line. The fidelity of deterministic replay must guarantee that the same *application* instructions and system calls are executed in the same order in all executions, but also allow replays to execute additional *instrumentation* instructions and system calls. These requirements preclude off-the-shelf use of any existing deterministic replay system. Some systems do not guarantee the same sequence of application instructions [3, 56], while others do not allow recorded and replayed executions to diverge sufficiently to run instrumentation code in one execution but not the other [74] or have unacceptably high recording overhead [51, 57]. Since ConfAid uses Pin binary instrumentation tool [44] for its analysis, we cannot use record and replay systems such as Aftersight [20] that perform their analysis in the VMM layer.

Our approach to solving this dilemma is co-design: we make the deterministic replay

system *instrumentation-aware* so that it compensates for the specific divergences in replayed execution caused by the dynamic instrumentation. Our replay system is designed to work with the Pin dynamic instrumentation tool. The replay code compensates for extra system calls made by Pin and the modifications to recorded system calls due to instrumentation. It also preallocates resources such as memory regions and signal handlers to avoid conflicts between the instrumentation and the replayed application. Instrumentation-awareness enables our replay system to provide the exact fidelity required by ConfAid. We describe the implementation details in the next section.

4.2 Implementation

Our deterministic record and replay system is implemented in the Linux kernel. The unit of replay can be either a single process or a group of communicating processes. Thus, our system records and replays one or more applications executing on the same computer.

Our system currently uses a standard design to record and replay processes. It take a checkpoint (address space and registers) of the process or processes being recorded. For each such process, our system logs the data returned by all system calls that the process executes. The logged values include addresses modified by the kernel within the process's address space. We also record the value and timing of signals delivered to each process. When recorded processes spawn child processes, we record the activities of the children — this is useful for servers that use children to handle incoming requests.

To replay a recorded execution, our system restarts the application from the checkpoint. When the application makes a system call, our kernel does not re-execute that call. Instead, it supplies the recorded values from the log of non-deterministic events. The exception to this rule is system calls such as `mmap` that change the address space of the application — such calls are executed by the replaying kernel in a manner that ensures that they produce an identical effect on the calling process's address space that was produced during recording. Our kernel also delivers the same signals to each process at the point the original signal

was received in the recorded execution. This guarantees high fidelity replay; i.e., that the recorded and replayed processes execute the same instructions and system calls in the same sequential order.

ConfAid analysis tool uses Pin to monitor information flow. While Pin is designed to be invisible to the application being instrumented, it is *not* designed to be transparent to lower layers of the system such as the operating system. For instance, Pin adds and modifies system calls, modifies signal handlers, and reserves memory addresses in the application address space.

ConfAid compensates for divergences in execution due to binary instrumentation. It allocates memory for use as a communication channel between the kernel replay system and the analysis tools run by Pin. The analysis tool uses this region to inform the kernel which system calls are initiated by the application (and hence should be replayed from the log) and which are initiated by Pin or the analysis tool (and should be executed normally). ConfAid intercepts all system calls issued by the application and sets a flag in this region prior to issuing the system call; it clears the flag when the system call ends. Thus, when the kernel sees a system call with the flag cleared, it knows that Pin or the analysis tool has issued the system call.

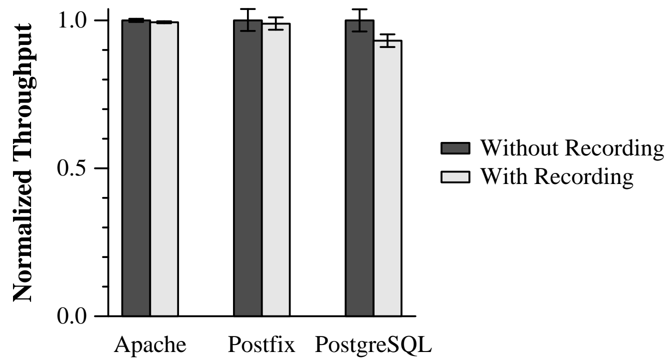
ConfAid also compensates for interference between system calls made by the recorded application and system calls made by Pin or the analysis tool. For instance, we observed that Pin would sometimes ask the kernel to `mmap` a free region of memory and the kernel would return a region that would later be requested by the recorded application, leading to a conflict. We compensated for this by scanning the log to identify all regions that will be requested by the recorded application during the replay and reserving these regions so that the kernel does not allocate them for Pin. We made similar modifications to compensate for conflicting requests for signal handlers and other resources that could potentially be requested by both the application and the dynamic instrumentation.

For inter-process communication, ConfAid originally transmitted taint values over the

same channels that were used to send the tainted data between processes. However, these channels do not exist during replay, since the kernel does not re-execute recorded system calls for inter-process communication. To solve this problem, ConfAid modifies the application binaries to establish and use special *side channels* (replay-specific TCP connections) for communicating taint values with each other. Since side channels are established by instrumentation and not by the application, the kernel executes side channel system calls during replay. During replay, when the instrumentation sees that one recorded process communicated with another, it uses the side channel to transmit the taint values from the sending process to the process that received the data during recording. The receiving process blocks until information is available on the side channel. This means that the replayed processes obey the same causal order of execution that they followed during recording.

We are currently modifying ConfAid to support multi-threaded applications. The biggest challenge has been supporting the needed fidelity of deterministic record and replay while adding low overhead to the production system. Several recent deterministic replay systems have lowered record overhead for multi-threaded processes running on multiprocessors by searching either online [69] or offline [3, 56] for a replayed execution that is equivalent only in external output to the recorded system. Like these prior systems, we plan to record system calls and user-level synchronization operations. During replay, we can enforce the same *happens-before* order among these operations that was observed during recording. In the absence of data races, this guarantees that the same sequence of instructions and system calls is executed by each pair of corresponding record/replay threads.

To deal with data races, we plan to run a dynamic data race detector during offline replay; we expect that the relative performance impact of this additional step will be small because we already execute high-overhead dynamic instrumentation during replay. During analysis, ConfAid will assign lower confidence to values accumulated from regions of code in which the executing thread is racing with another thread. The range of the potential error can be estimated by sampling different interleavings of racing instructions during replay.



This figure compares server throughput with and without our deterministic record system. Results are normalized to the number of requests per second without our recording. Higher values are better. Each result is the mean of 10 trials; error bars are 95% confidence intervals.

Figure 4.1: Overhead of deterministic recording

ConfAid users can either use the lower-confidence results, or they can add annotation or synchronization to the application to eliminate the data races.

4.3 Evaluation

We used our deterministic record and replay system to record three applications: the Apache Web server version 2.2.14, the Postfix mail server version 2.7 and the PostgreSQL database version 9.0.4. We ran all experiments on a Dell OptiPlex 980 with a 3.47 GHz Intel Core i5 Dual Core processor and 4 GB of memory, running a Linux 2.6.26 kernel modified to support deterministic record and replay. We measured online overhead by comparing the throughput and the latency of these three applications when they are recorded by our record system to the results of running the applications on the default Linux kernel without recording.

Figure 4.1 shows that our recording adds a 1–7% throughput overhead for the three applications. For Apache, we used `ab` to send 5000 requests for a 35 KB static Web page with a concurrency of 50 requests at a time over an isolated network. Our recording reduced throughput by 0.6%. Per-request latency increased by 0.6%. The recording log size for this

experiment was 7 MB, containing 115K system calls.

For Postfix, we used the `smtp-source` tool to send 10000 mail messages of size 1 KB from another machine on the isolated network. Postfix processing is asynchronous, so there is no meaningful latency measure. Our recording reduced server throughput by 1.1%. The log size was 453 MB, containing 6 million system calls.

We benchmarked PostgreSQL using `pgbench`. We measured the number of transactions completed in 60 seconds with concurrency of 10 transactions sent at a time. Each transaction involves one `SELECT`, three `UPDATE`s, and one `INSERT` command. Our recording reduced throughput by 7% and increased per-request latency by 7%. The log size was 820 MB, containing 17 million system calls. We conjecture that the higher overhead for PostgreSQL was mostly due to the increased log size and larger number of executed system calls.

CHAPTER V

X-ray: Troubleshooting Performance Anomalies with Causality Analysis

5.1 Introduction

Understanding and troubleshooting performance problems in complex software systems is notoriously challenging. This challenge is compounded for software in production for several reasons. To avoid slowing down production systems, analysis and troubleshooting must incur minimal overhead. Further, performance issues in production can be both rare and non-deterministic, making the issues hard to reproduce.

However, we argue that the most important reason why troubleshooting performance in production systems is challenging is that current tools only solve half the problem. Troubleshooting a performance anomaly is essentially the process of determining *why* certain events, such as high latency or resource usage, happened in a system. Unfortunately, most current analysis tools, such as profilers and logging, only determine *what* events happened during a performance anomaly — they leave the more challenging question of why those events happened unanswered. Administrators and developers must manually infer the root cause of performance issue from the observed events based upon their expertise and knowledge of the software. For instance, a logging tool may detect that a certain low-level routine is called often during periods of high request latency, but the user of the tool must then infer

that the routine is called more often due to a specific configuration setting.

The final part of this thesis introduces the technique of *performance summarization* which not only determines what events occurred during a performance anomaly, but also determines why the anomaly occurred. Performance summarization first attributes performance costs such as latency and I/O utilization to fine-grained events (individual instructions and system calls). Then, it uses fine-grained causality analysis, similar to ConfAid, to associate each such event with a set of probable root causes such as configuration settings or specific data from input requests. The cost of each event is assigned to potential root causes weighted by the probability that the particular root cause led to the execution of that event. Finally, the per-cause costs for all events in the program execution are summed together. The end result is a list of root causes ordered by their performance costs. In the above example, the outcome of performance summarization would indicate that one specific configuration setting contributed the most to the performance slowdown. This output gives the system troubleshooter a direct indication of how to fix the problem, without the need for time-consuming manual analysis.

We also introduce *differential performance analysis* which is used to determine why the performance impact of two different events differed. For instance, differential performance analysis can be used to understand why two requests to a Web server took different amounts of time to complete. Differential performance analysis identifies branches where the execution paths of the two requests diverged. It assigns a performance cost to each path taken from the branch, then uses dynamic information flow analysis to determine why the two requests diverged at that point. It attributes the difference in performance costs between the two paths to the identified root causes according to the likelihood that they caused the branch condition to evaluate to different values during the two executions. The costs of all such divergences during are summed. The output shows the system troubleshooter a set of reasons why the performance costs of two requests differ, along with a specific performance impact for each reason.

We have built a tool called X-ray that implements performance summarization. X-ray attributes latency, CPU utilization, file system usage, and network utilization to specific root causes. X-ray supports several different scopes of analysis: intervals of time, specific requests, or a differential analysis of pairs of requests. Thus, X-ray can answer performance questions such as:

- Why did a particular request take a long time to execute?
- Why is file system usage high during a specific time period?
- Why did request R take longer to execute than request S?

X-ray leverages the deterministic record and replay system introduced in chapter IV to offload the heavy-weight root cause analysis from the production system. As explained in chapter IV, a deterministic replay system provides DVR-like functionality, in which an execution of a hardware or software system is recorded so that an identical execution can later be replayed on demand. For the purpose of X-ray, we slightly modified our deterministic record system to capture all the performance-related information online, in addition to other non-deterministic events. For example, X-ray collects the timing information during recording, because the offline heavyweight analysis substantially perturbs timing. During replay, X-ray determines the root causes of the execution of each event and associates the collected performance costs to those root causes.

Thus, the contributions of this part of my thesis are the following:

- The technique of performance summarization, which attributes performance costs to root causes.
- The technique of differential performance summarization for understanding why two similar events have different performance.
- Development and evaluation of the X-ray tool, which implements these techniques.

We evaluated X-ray using three applications: the Apache Web server, the Postfix mail server and the PostgreSQL database. We have reproduced and analyzed 14 performance issues reported for these applications. In 12 of 14 cases, X-ray identifies a correct root cause as the largest contributor to the performance problem; in the remaining two cases, X-ray identifies a correct root cause as the third largest contributor.

5.2 X-ray overview

5.2.1 Troubleshooting with X-ray

X-ray pinpoints why performance anomalies, such as high request latencies or bottlenecks in resources, occurred on a production system. Our current system targets servers, though this is not fundamental to our design.

X-ray does not require application source code because its analysis operates entirely on application binaries and modifications are made using dynamic binary instrumentation. Thus, X-ray can be used on COTS (common off-the-shelf) applications, making the tool appropriate for system administrators as well as for developers.

The first step in using X-ray is to record an interval of software execution on a production system. X-ray uses our deterministic record and replay system, introduced in chapter IV. As we showed earlier, our recording overhead is currently only 1–7%. Thus, a user can choose to leave the record system running for long periods of time to capture rare and hard-to-reproduce performance issues. Alternatively, the record system can be dynamically enabled only when specific performance issues are exhibited.

X-ray performs its analysis offline on the replayed execution. An X-ray user chooses which interval of execution to analyze. The user may select the entire execution, an interval of time, or a specific input request. X-ray produces a performance summary for the selected interval. The first two intervals are appropriate when the user notices degraded throughput over a period of time, whereas the latter is best when one or more requests take

```
Allow domain.name (line 164) : 603 usecs
ServerRoot (line 29)         : 151 usecs
TypesConfig (line 298)       : 151 usecs
<IfModule(line 231)          : 75 usecs
alias\_module(line 231)      : 75 usecs
<Directory(line 162)         : 55 usecs
...
```

Figure 5.1: Example of X-ray output for Apache

an unexpected amount of time to execute. Alternatively, a user may select two requests to compare, in which case X-ray does a differential performance summarization for the selected requests. Typically, a user would select two similar requests that differ substantially in service time, though our results show that X-ray will provide useful information even when the two selected requests are very dissimilar.

The X-ray user next selects the set of performance statistics to summarize. Typically, we expect that a user will use basic performance analysis tools such as `top` and `iostat` to identify the bottleneck resource. X-ray provides a flexible framework for analyzing arbitrary statistics; our current implementation supports latency, CPU utilization, file system usage, and network bandwidth.

Figure 5.1 shows an example of X-ray output for Apache. The output shows the inferred root causes of a performance problem. X-ray associates a specific cost (in this case, latency) to each root cause and orders the list by that metric. In the figure, all root causes are from the `httpd.conf` configuration file. Based on X-ray output, users may identify configuration options that are inappropriate for their workload, they might choose a set of configuration options that offer a different tradeoff between performance or functionality, or they may reprovision their system to supply resources in quantities that match the features they desire.

The recorded executions can be replayed multiple times. Therefore, X-ray users can perform many different analyses for the same recording. For instance, a user may change the scope of execution analyzed, choose different metrics to summarize, or switch between basic and differential performance summarization. This means that the X-ray user does

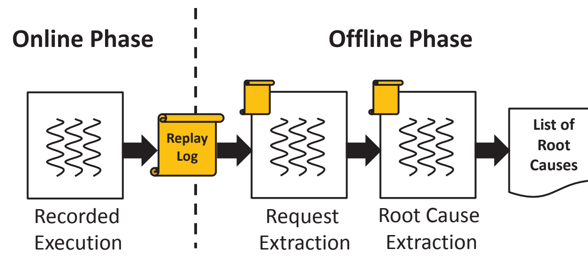


Figure 5.2: Overview of X-ray

not need to decide what type of analysis will be useful before a performance anomaly is recorded.

5.2.2 Mechanics of X-ray

Figure 5.2 shows an overview of how X-ray runs. X-ray divides its analysis between the recorded and replayed executions. In the online phase, along with recording system calls and other non-deterministic events, X-ray also records timing information and other performance-specific data because the subsequent, offline analysis perturbs the execution too much to accurately measure performance.

In its offline phase, X-ray performs two passes, each of which is a deterministic replay of the recorded execution. In the first pass, X-ray performs *request extraction*, in which it determines the specific intervals of execution (i.e., the basic blocks executed) during which each process is handling each input request to the recorded system. In the first pass, X-ray also assigns the recorded performance costs to each instruction and system call. In the second pass, X-ray completes performance summarization by using dynamic information flow to attribute events to root causes and by calculating the cost of each root cause. At the end of the second pass, X-ray outputs a list of root causes ordered by its user's chosen performance metric.

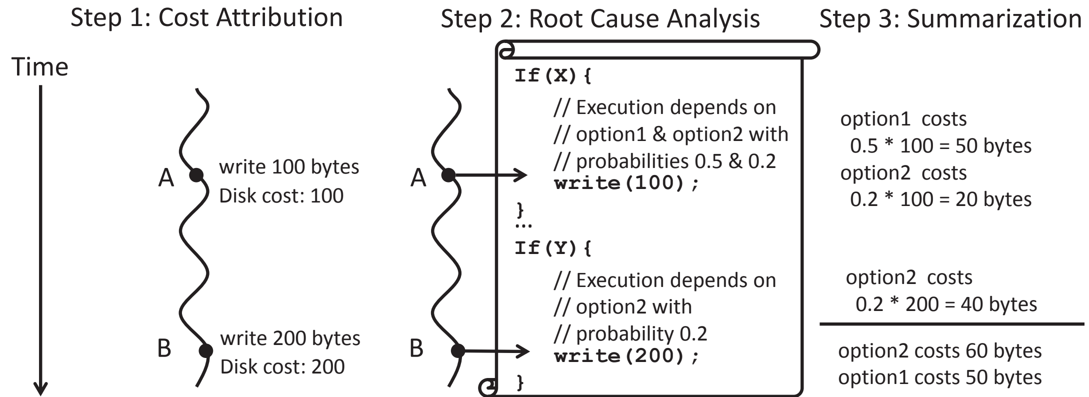


Figure 5.3: Example of performance summarization

5.3 Performance summarization

Performance summarization is the heart of X-ray. The goal is to attribute specific performance costs such as request latency, CPU usage, and I/O utilization to one or more root causes. X-ray considers any configuration option or any data received from an input request as a potential root cause.

5.3.1 Basic performance summarization

Performance summarization is akin to integration in calculus. X-ray individually analyzes the per-cause performance cost and root cause of each user-level instruction and system call (referred to as events in the discussion below), then adds together the per-event costs to calculate how much each root cause has reflected the performance of the application during the period of observation selected by the X-ray user.

Figure 5.3 shows an overview of how performance summarization works. In the first step, X-ray attributes performance metrics to each event executed by one or more processes comprising a server application; the figure assumes that the X-ray user has specified file system usage as a metric. Some metrics such as file system utilization are associated only with system calls, while others such as latency are attributed to both system calls and user-level instructions.

In the next step, X-ray uses dynamic information flow analysis to derive a set of possible root causes for the execution of each event. Essentially, this step answers the question: ”how likely is it that changing a configuration option or receiving a different input would have prevented this event from executing?” X-ray uses the same algorithms that we developed in ConfAid to perform this analysis. In the last step, X-ray multiplies the performance metrics for each event by the per-cause taint values to derive the per-event performance cost for each root cause. X-ray sums these costs over all events that executed during the period selected by the user and outputs an ordered list of root causes.

5.3.2 Differential performance summarization

Differential performance summarization is a technique for comparing any two executions of an application activity, such as the processing of two different request by a Web server. Such activities have a common starting point (e.g., the receipt of a request) and termination point (e.g., the sending of a response), but the execution paths for different events may diverge due to differences in the input or specific configuration settings.

Figure 5.4 shows an example of differential performance summarization. X-ray compares two activities by first identifying all points where the paths of the two executions diverge. It then uses causality analysis to evaluate why each divergence occurred; this reason is given by the taint of the branch conditional at the divergence point. For each performance metric, X-ray calculates the cost of the divergence by subtracting the cost of all events on the divergent path taken by the first execution from the cost of all events on the path taken by the second execution. This cost is attributed to root causes by multiplying the metric values by the taint weight. X-ray sums the per-cause costs of all divergences and output a list of root causes ordered by the differential cost.

5.4 Implementation

We next describe the implementation of X-ray in detail.

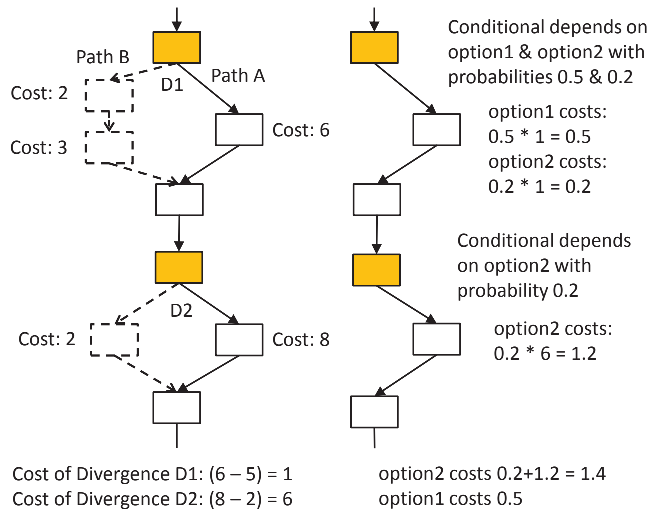


Figure 5.4: Example of differential performance summarization

5.4.1 Online phase

Since the online phase of X-ray analysis runs on a production system, X-ray uses deterministic record and replay to move any activity with substantial performance overhead to a subsequent, offline phase. The only two activities performed online are recording non-deterministic inputs and gathering performance information.

5.4.1.1 Recording performance information

Since X-ray analysis imposes a runtime overhead of several orders of magnitude, timing information gathered during an instrumented run is essentially useless for diagnosing most performance problems. In contrast, timing information gathered during the recorded run captures the exact performance experienced by the production system. X-ray therefore gathers timing data during recoding and explains the timing data by reasoning about the instructions and system calls executed during replayed executions.

To capture timing information, for each system call executed by the application, the kernel records the system time at kernel entry and exit. For simplicity, the kernel writes the timing information for each system call to the same log that it uses to store non-

deterministic events. Analysis tools read the log directly to extract the timing information during replay. Other performance information, such as the number of bytes read or written during I/O system calls are already captured as a result of recording sources of non-determinism.

5.4.2 Offline phase

X-ray executes analysis in two passes. In the first pass, X-ray performs request extraction to determine when each application process is handling each request. It also identifies which basic blocks are executed within the analysis scope chosen by the user and attributes performance costs to those blocks. In the second pass, X-ray attributes basic block execution to specific root causes and summarizes the performance cost for each cause. Since X-ray operates on a previously-recorded execution, it is trivial to replay the execution multiple times so that different parts of the analysis can be executed sequentially (much like a multi-pass compiler).

5.4.2.1 Request extraction

During the request extraction phase, X-ray identifies the intervals of application execution during which each request was processed. For many types of analysis, X-ray must understand how an application processes one or more particular requests such as particular mail messages for the Postfix mail server or Web requests for Apache. Request extraction traces the causal path of each request from the point when the request is received by the application to the point when the request terminates (e.g., when a server sends the response). Often, requests traverse multiple processes, and different processes handle different requests at the same time.

The notion of a request is application-dependent. Thus, X-ray requires a per-application filter that specifies the boundaries of incoming requests. The filter is simply a regular expression over incoming data. For instance, the Postfix filter looks for the string `HELO`

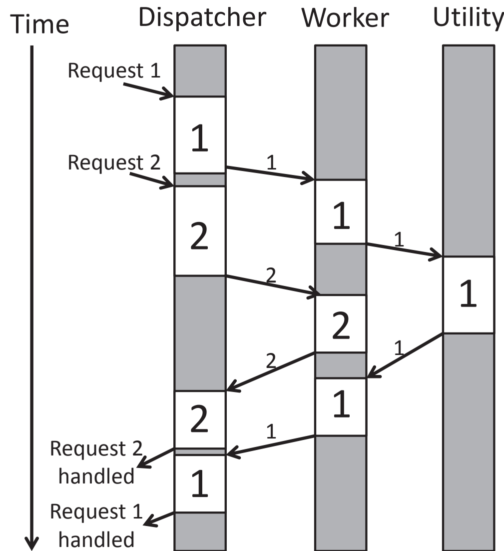


Figure 5.5: An example of X-ray request extraction. The intervals marked as 1 or 2 in each process correspond to the portions of process execution that X-ray associates with the first and second requests, respectively.

to identify incoming mails. A filter only needs to be created once for each protocol (e.g., SMTP or HTTP).

Request extraction runs as a Pin tool. The tool examines values returned from all system calls that provide external input such as those that receive data from the network. When the data returned from such system calls match the specified filter, X-ray tags the receiving process with a unique request identifier to show that it is handling the request in question.

As shown in Figure 5.5, X-ray propagates request tags among processes as they communicate. It currently assumes that each process handles a single request at a time, but it allows multiple processes to concurrently handle different requests (for instance, the dispatcher handles request 2 while a worker handles request 1 in the figure). When a message with a new tag is received by a process, X-ray assumes that it ceases to handle the old request and starts to handle the new one. This assumption is valid for the server applications we use in the evaluation.

Note that since these processes are being replayed, the kernel does not actually send and receive data when they execute system calls. Therefore, request extraction cannot use

existing communication channels to propagate request tags. We therefore create and use side channels, as described in chapter IV, to communicate request tags between the sending and receiving processes.

Although most popular servers such as Apache, Postfix or PostgreSQL handle a single request per thread of execution, event-based servers may handle many requests simultaneously using a single thread. Since X-ray already tracks application data flow, we plan to extend X-ray to handle such servers via fine-grained information flow analysis (i.e., taint tracking). Essentially, we can identify the memory addresses associated with each request and use that information to identify the code intervals in which a thread or process is handling a particular request. Alternatively, we could use per-application schemas as is done during Magpie request extraction [5].

As the replayed application processes execute, the request extraction Pin tool tags each basic block with a request identifier if it believes the process is handling a request at that time. The final output of the request-extraction instrumentation is a per-request list of <process,basic block> tuples in the order that the basic blocks were executed.

5.4.2.2 Identifying basic blocks

The first step in performance summarization is to map the scope of the analysis specified by the user to a set of basic blocks. If the user specifies the scope as a time interval, X-ray includes all basic blocks executed by any process within that interval. Identification is somewhat imprecise because X-ray only records timestamps at the entry and exit of system calls. The analyzed scope is from the exit of the last system call executed before the specified interval to the entry of the first system call executed after the specified interval. If the analysis scope is a time interval, X-ray omits request extraction because it is not needed.

If the user specifies a particular request as the scope of analysis, X-ray uses the request extraction results that identify the set of basic blocks for that request. If the user specifies

two requests to compare using differential performance analysis, X-ray uses the request extraction results for both requests.

5.4.2.3 Attributing performance costs

X-ray next attributes specific performance costs to events (application instructions and system calls executed). As a performance optimization, X-ray considers all events in the same basic block together since they have the same set of root causes (in other words, if one event is executed, they all must be executed).

Currently, users may choose one or more of the following metrics: latency, CPU utilization, file system usage, and network throughput. During recording, X-ray records the start and end time of every system call in the log of non-deterministic events. When it encounters the same system call during replay, the Pin tool reads the log and subtracts the two values to determine the system call latency. The latency is then attributed to the basic block that invoked the system call.

X-ray next considers latency not attributable to system calls. It currently uses a simple method that attributes latency in proportion to the number of user-level instructions executed. X-ray then takes the total process execution time, subtracts the time spent in system calls, and divides the remaining time by the number of instructions. The result is the latency per instruction. Multiplying this value by the number of instructions in a basic block and adding in any system call latency for that block gives the block's total latency.

To calculate CPU utilization, X-ray simply counts the number of instructions executed by each basic block. To calculate file system and network usage, it inspects the replay log as it is replayed to identify file descriptors associated with the resource being analyzed. When a system call reads or writes data for these descriptors, X-ray attributes the total number of bytes processed to the basic block that invoked the system call.

5.4.2.4 Information flow analysis

X-ray next determines why each basic block executed. X-ray uses ConfAid to generate a set of probable root causes for each block. ConfAid assigns a unique taint identifier to registers and memory addresses when data is read into the program from configuration files and incoming request sockets. It identifies specific configuration tokens through a simple form of symbolic execution. For instance, if data read from a known configuration file is compared to the string “FOO”, then ConfAid marks that data as associated with token `FOO`.

As the program executes, ConfAid propagates taint identifiers to other locations in the process’s address space according to dependencies introduced via data and control flow. Rather than track taint as a binary value, it associates a weight with each taint identifier that represents the strength of the causal relationship between the tainted value and the root cause. X-ray builds on ConfAid by also assigning a weighted set of taint values to each basic block that is executed; membership in this set indicates that the block’s execution depends on the specified root cause, and the associated weight indicates the strength of the dependency.

We modified ConfAid to better suit the needs of X-ray. Our first modification was to broaden the source of tainted data. X-ray not only tracks data read from configuration sources; it also tracks data read from input requests. X-ray uses the same filter that it uses during request extraction to determine when the application is reading data from a request. The taint identifier in this case indicates the particular request on which a memory address or register depends.

We also modified how ConfAid uses taint values. The original ConfAid implementation only outputs taint values when it encounters an application failure. However, X-ray is interested in the taint values of all instructions and system calls executed within the scope of analysis. During execution, our modified version of ConfAid generates a taint set that contains root causes and associated weights for every basic block that has been marked as being within the scope of analysis.

As an example, our modified version of ConfAid might emit the following taint set for a basic block: {`FOO` : `1.0`, `BAR` : `0.5`}. This represents the belief that the basic block would definitely not have been executed if root cause `FOO` were different and the belief that the block is 50% likely not to have been executed if root cause `BAR` were different. Note that these are two independent probabilities: potentially changing either of the two options might cause the basic block to not have been executed. Thus, the values in a taint set need not sum to one.

5.4.2.5 Integration

Next, X-ray attributes the performance cost of executing each basic block according to specific root causes. For each root cause in the block's taint set, X-ray multiplies the per-block cost by the weight associated with the root cause. Each process maintains a running sum of the costs associated with each root cause as it is replayed. The final cost for each root cause is determined by adding together the sums from all replayed processes. At the end of analysis, X-ray prints out a list of root causes and shows the estimated performance cost for each. X-ray can simultaneously analyze multiple performance metrics.

5.4.2.6 Differential performance summarization

X-ray uses a different method to compare the performance of two requests. It first identifies the points where the execution paths diverged from one another. It uses the results of request extraction to output each path as a sequence of basic blocks executed by the request. Each path may span multiple processes. X-ray then uses the `diff` tool to compare the two paths and understand where they diverged from one another and where the divergence ended as the paths merged back together.

X-ray then determines the root cause of each divergence. It attributes the cost of the divergence to the conditional that immediately preceded the divergence. It calculates a performance cost for the divergence by first summing the performance costs of all basic blocks

along the divergent path for one request and then subtracting the sum of the performance costs of all basic blocks along the divergent path for the other request. It attributes the divergence to root causes by multiplying the cost of the divergence by the weights in the taint set for the conditional that caused the divergence.

5.5 Evaluation

Our evaluation of X-ray answers the following questions:

- How accurately does X-ray identify the root cause of performance problems?
- How fast can X-ray troubleshoot a performance problem?

5.5.1 Experimental Setup

We used X-ray to diagnose performance problems in three applications: the Apache Web server version 2.2.14, the Postfix mail server version 2.7 and the PostgreSQL database version 9.0.4. In Apache, each request is handled by one process. Postfix has multiple utility processes, each of which handles a certain part of a request. On average, a Postfix request is handled by 5 different processes. In PostgreSQL, each request is handled by one process. However, PostgreSQL has multiple time-based utility processes such as a write-ahead log writer and an auto-vacuum that handle requests in batches. We ran all experiments on a Dell OptiPlex 980 with a 3.47 GHz Intel Core i5 Dual Core processor and 4 GB of memory, running a Linux 2.6.26 kernel modified to support deterministic replay.

5.5.2 Root cause identification

We evaluated X-ray by recreating known performance issues reported in application performance tuning and troubleshooting Web pages, forums, and blog posts. To recreate each issue, we either modified configuration settings or sent a problematic sequence of

App	#	Description of performance test cases
Apache	1,2	Apache sets a threshold for the number of requests that are handled in one TCP connection using the KeepAlive and MaxKeepAliveRequests setting. A low threshold causes Apache to shut down and rebuild the connections too often, causing a significant delay in handling some requests.
	3	In Apache, access to various directories can be controlled in the config file based on the domain name of the client sending the request. This setting causes extra DNS calls for verifying the domains and leads to high latency in handling the requests.
	4	Apache can be configured to log the host names of clients sending requests to specific directories for administrative purposes. This setting causes extra DNS calls and leads to high latencies in handling requests for those directories.
	5	Apache can be configured to require authentication for some directories. Authentication causes high CPU usage peaks.
	6	Apache can be configured to generate content-MD5 headers calculated using the message body. This header provides an end-to-end message integrity with high confidence. However, for larger files, the calculation of the digests causes high CPU usage.
	7	By default, Apache sends eTags in the header of HTTP responses. The eTags can be used by the client in future requests for the same file to only receive the file if its contents have changed.
Postfix	1	Postfix can be enabled to log more information for a list of specific hosts, using debug_peer_list option. The extra logging causes excessive disk activity.
	2	Postfix can be configured to examine the body of the messages against a list of regular expressions known to be from spammers or viruses. This setting can significantly increase the CPU usage for handling a received message if there are many expression patterns.
	3	Postfix can be configured to reject requests that are sent from blacklisted domains. Postfix uses DNS mechanism to query blacklist operators to determine if the message should be rejected. Based on the number of operators specified, Postfix performs extra DNS calls, which significantly increases the latency of the handled message.
PostgreSQL	1	PostgreSQL tries to identify the correct time zone of the system for displaying and interpreting time stamps if the time zone is not specified in the configuration file. This increases the startup time of PostgreSQL by 5x.
	2	PostgreSQL can be configured to synchronously commit the write-ahead logs to disk before sending the end of the transaction message to the client. This setting can cause extra delays in processing transactions if the system is under a large load.
	3	The frequency of taking checkpoints from the write-ahead log can be configured in the PostgreSQL configuration file. Having more frequent checkpoints decreases crash recovery time but significantly increases disk activity for busy databases.
	4	The delay between the activity rounds of the write-ahead log write process can be configured in PostgreSQL configuration file. Setting this delay higher causes potential loss of transactions. However, lower delays cause extra CPU usage.

Table 5.1: Description of the Apache, Postfix and PostgreSQL performance test cases

requests to the server. In total, we recreated the 14 problems described in Table 5.1 (7 for Apache, 3 for Postfix, and 4 for PostgreSQL).

For each test case, we recorded server execution while we sent several application requests. We used standard lightweight performance monitoring tools such as top, iostat, netstat and log files to identify the bottleneck resource and identify requests during which resource usage was high. Later, we executed X-ray offline analysis of the recorded runs to explain the performance anomalies.

For each test case, Table 5.2 shows the scope and metric we used for X-ray analysis. The next column shows the top three root causes identified by X-ray, along with X-ray's analysis of how much the cause contributed to the performance metric under observation. The correct answers for each test case is shown in bold. The last column shows how long X-ray offline analysis took.

5.5.2.1 Apache

In the first Apache test case, the threshold for the number of requests that can reuse the same TCP connection is set too low. Re-establishing a connection causes some requests to exhibit higher latency than others. To exhibit this problem, we sent 100 various requests to the Apache server using the *ab* Apache benchmarking tool. The requests used different HTTP methods (GET and POST) and asked for files with different sizes.

We first used X-ray to perform a differential performance summarization of two similar requests (HTTP GETs of small files), one of which had a small latency and one of which had a high latency. X-ray correctly identified the `MaxKeepAliveRequests` token as the largest root cause.

Next, we explored how sensitive X-ray is to the similarity of the compared requests (Apache test case 2). We compared two very dissimilar requests using differential performance summarization: a small HTTP POST and a large HTTP GET. As would be expected, X-ray reported that the largest cause of the divergence in processing time was due to the input data from the requests. The `DocumentRoot` parameter is also reported as a large cause of the divergence because the root is appended to the input file name. However, X-ray

still reported that the `MaxKeepAliveRequests` is a substantial reason for divergence. Further, the estimated performance impact of `MaxKeepAliveRequests` is not affected much by the similarity of the requests.

This test case highlights the power of differential performance summarization. X-ray does not require two requests to be substantially similar in order to identify performance anomalies. Because it analyzes program control flow, X-ray can correctly differentiate performance differences due to diverging input from those due to other root causes such as configuration options.

In the third test case, Apache is configured to use the `Allow` directive with a domain name to control access to a certain directory. Apache performs two DNS calls to determine the domain names of clients. These extra DNS calls increase the latency for requests that access the directory with the domain-name access control. In this test case, we used X-ray differential analysis to compare the latencies of two requests: one accessing a directory with domain-name access control, and one accessing a directory with no access control. X-ray correctly attributed the high latency to the `Allow` directive.

Apache can be configured to log the host names of clients that send requests to particular directories. This setting can be turned on using the `HostNameLookups` directive, and is mostly used for administrative purposes. To determine the host names, Apache performs extra DNS calls, which lead to high latencies when handling requests for directories with enabled logging option. In the fourth test case, we used X-ray to compare the latencies of two requests: one accessing a directory with logging option, and one accessing a directory without any host name logging. X-ray was able to correctly identify the `HostNameLookups On` setting as the biggest contributor to the latency problem.

In the fifth test case, we configured Apache to require authentication for a certain directory. We used the `AuthUserFile` option to specify the file that contains the list of allowed usernames along with their encrypted passwords. When a request accesses the directory with enabled authentication, the system experiences a high CPU usage because

Apache executes CPU-intensive encryption functions. We used X-ray to analyze the high CPU usage for a request accessing that directory. X-ray correctly identified the authentication option as the biggest root cause of CPU usage.

Apache can be configured to calculate an MD5 message digest for a request. This digest can be used as a fingerprint to verify end-to-end message integrity. However, the digest calculation can cause high CPU usage for large files. In the sixth test case, we used X-ray differential analysis to compare the CPU usage of two requests. The first request accesses a directory for which Apache generates a message digest, while the second request does not require one. X-ray identified the `ContentDigest` option as the biggest contributor to the difference between the CPU usage of the two requests.

In the last Apache test case, the root cause of high network usage is the client's failure to use the HTTP conditional eTag header. A recent study [60] found that many smartphone HTTP libraries do not support this option, causing redundant network traffic. X-ray identifies this problem via differential analysis, showing that it can sometimes identify bad client behavior via analysis of servers. We verified that correct eTag support substantially reduces network load.

X-ray analysis time for the 7 test cases varies between 2 and 3 minutes. This is very reasonable considering that analysis is performed offline and does not affect the online production software.

5.5.2.2 Postfix

The first Postfix test case reproduces a problem reported in a Postfix user's blog [58]. The user noticed that emails with attachments sent from his account transferred very slowly, while everything else, including the mail received by IMAP services, had no performance issues.

The user employed `iotop` to monitor the Postfix server, and observed that one child process was generating a lot of file system activity. He poured through the server logs and

App	#	Scope & Metric	Results : Expected contribution	time
Apache	1	Diff, Latency	**MaxKeepAliveRequests**: 17.2 usecs. KeepAlive On: 8.6 usecs. Directory: 4.7 usecs.	2m 40s
	2	Diff, Latency	**User's request**: 311.6 usecs. **DocumentRoot**: 311.5 usecs. **MaxKeepAliveRequests**: 16.8 usecs.	2m 41s
	3	Diff, Latency	**Allow domain.com**: 603 usecs. ServerRoot: 151 usecs. TypesConfig : 151uses	2m 14s
	4	Diff, Latency	**HostNameLookups On**: 254 usecs. Directory: 127 usecs. **HostNameLookups**: 127 usecs.	2m 4s
	5	Request, CPU	**AuthUserFile**: 9M instrs. User's request: 600K instrs. Listen: 80K instrs.	2m 6s
	6	Diff, CPU	**ContentDigest On**: 217K instrs. **ContentDigest**: 108K instrs. Directory: 108K instrs.	2m 6s
	7	Diff, Network	**User's request**: 35 KB **DocumentRoot**: 35 KB Listen: 4 KB	2m 4s
Postfix	1	Request, File system	**User's request**: 100 KB **debug_peer_list**: 28 KB queue_directory: 5 KB	1m 18s
	2	Request, CPU	**body_checks**: 1M instrs. **User's request**: 900K instrs. myhostname: 300K instrs.	2m 49s
	3	Request, Latency	**reject_rbl_client**: 3.5 secs. **reject_rbl_client**: 1.9 secs. **smtpd_client_restrictions**: 0.9 secs.	1m 24s
PostgreSQL	1	Time int., CPU	**timezone**: 28M instrs. default_text_search_config: 11M instrs. datestyle: 11M instrs.	15+m
	2	Request, Latency	shared_buffers: 0.42 secs. max_connections: 0.26 secs. **wal_sync_method**: 0.26 secs.	2m 50s
	3	Time int., File system	**checkpoint_timeout**: 16 KB shared_buffers: 11 KB max_connections: 11 KB	4m 48s
	4	Time int., CPU	shared_buffers: 2.6M instrs. max_connections: 2M instrs. **wal_writer_delay**: 1.4M instrs.	5m 27s

Table 5.2: The results for our performance test cases.

realized that the child process was logging large amounts of data. Finally, he ran through his configuration file, and eventually found out that the `debug_peer_list`, which specifies

a list of hosts that triggered the logging, included his own IP address.

Our results show that X-ray can make this diagnosis automatically. We simply analyzed a specific request that was associated with a period of high file system usage. X-ray identifies both the request (since it contains the IP address that caused excessive logging) and the erroneous parameter as the top two root causes, pinpointing the specific reasons for the high file system activity. Note that we did not have to identify which child process was responsible for the logging, nor did we have to read any log files. Since X-ray produced these results in a little over a minute, our tool could have saved the blogger considerable time.

In the second test case, Postfix is configured to perform spam filtering by comparing the body of the email message against a list of regular expressions known to be from spammers or viruses. If there are many patterns, the regular expression matching can significantly increase the CPU usage, when Postfix is handling an incoming email. In this test case, we used X-ray to analyze the CPU utilization of a single request. X-ray was able to correctly identify the `body_checks` option as the root cause of high CPU usage.

In the last test case, we configured Postfix to identify and reject requests that come from blacklisted domains. Postfix uses DNS mechanism to query blacklist operators and determine whether a request has come from a bad source. These DNS calls significantly increase the latency of request handling. In this test case, X-ray identifies the `reject_rbl_client` and the `smtpd_client_restrictions` options as the dominant sources of the latency.

As shown in Table 5.2, X-ray identifies the correct root cause for each Postfix problem in only a few minutes.

5.5.2.3 PostgreSQL

The first PostgreSQL case study is based on our own experience. Our evaluation started and stopped PostgreSQL many times. We noticed that our scripts were running slowly due

to application start-up delay, and decided to try to use X-ray to improve performance. Since top showed 100% CPU usage, we performed a X-ray CPU analysis during the interval before PostgreSQL received the first request.

Unexpectedly, X-ray identified the timezone configuration option as the top root cause. In the configuration file, we had set the timezone option to unknown. This caused PostgreSQL to expend a surprising amount of effort to attempt to identify the correct time zone. We updated the configuration to specify our time zone, and were pleased to see that the application startup time decreased by over 80%. While this problem is admittedly esoteric since most PostgreSQL users will not start and stop the application several times in succession, we were happy to see that X-ray could help identify performance issues that we did not specifically inject into the application.

PostgreSQL can be configured to synchronously commit transactions to disk before sending the end of the transaction message to the client. This option increases the latency of handling transactions, but it provides stronger guaranteers to the client by ensuring that the transaction is safely written to disk. In the second test case, we configured PostgreSQL with this option, and analyzed the latency of a single request. X-ray identified the correct configuration option as the third biggest contributor to the latency of the request. The shared_buffers and max_connections parameters appear to taint many branches during PostgreSQL execution causing them to rank as the first and second causes of latency.

Since PostgreSQL utility processes are mostly asynchronous (they sleep for a while and then wake up to perform tasks such as flushing write-ahead log to disk, taking checkpoints, or vacuuming the database) time interval analysis is a great fit for this application. In the third test case, we configured PostgreSQL to take checkpoints from the write-ahead log more frequently. This setting decreases crash recovery time, but increases file system activities. In this test case, we reduced the value of checkpoint_timeout option to increase the frequency of checkpoints, and used X-ray to analyze the file system activities over a time interval. X-ray was able to correctly associate the high file system activities to

the `checkpoint_timeout` option.

In the fourth PostgreSQL test case, we configured the activity rounds of the write-ahead log write process using the `wal_writer_delay` option. Choosing high writer delays increases the risk of losing transactions, and choosing low delays increases the CPU usage due to extra activities of the writer process. The fourth PostgreSQL test case analyzes the effects of this option by using X-ray time-interval analysis for CPU utilization. X-ray ranks the correct root cause third for this test case, after the `shared_buffer` and the `max_connections` options.

X-ray analysis time is currently capped at 15 minutes; analysis of the first test case hit this limit but still returned meaningful results since the analysis executed almost all the code used during startup. The remaining PostgreSQL issues required 2–5 minutes to analyze. We have not yet put much effort into optimizing X-ray analysis performance, since these times are still substantially faster than manual performance debugging.

5.6 Conclusion

Diagnosing performance problems in production systems is challenging. X-ray helps system administrators by identifying the root cause of observed performance problems. X-ray first records the execution of the production system and collects performance information. In an offline phase, X-ray deterministically replays the recorded execution and performs heavyweight causality analysis. X-ray uses dynamic information flow analysis to attribute the recorded performance information to root causes that include configuration options and request inputs. Our results show that X-ray accurately identifies the root cause of several real-world performance problems, while imposing only 1–7% overhead on a production system.

CHAPTER VI

Related Work

Several prior research efforts have applied different techniques to the problem of configuration troubleshooting.

PeerPressure [71] and its predecessor Strider [72] use statistical methods to compare configuration state in the Windows registry on different machines. When a value on a machine exhibiting erroneous behavior differs from the value usually chosen by other machines, PeerPressure flags the value as a potential error.

Similar to SigConf, PeerPressure and Strider employ a black-box approach towards troubleshooting and leverage the help of other execution states. However, PeerPressure and Strider benefit from the known schema of the registry and cannot detect configuration errors that lie outside the registry. The SigConf approach is more general and holds promise for dealing with errors that lie outside the registry and on other operating systems such as Unix variants. SigConf, however, assumes that the bug is already known and recorded in the reference computer, but PeerPressure and Strider do not have this assumption.

The downside of the black-box approach of PeerPressure and Strider is that it works well as long as the majority configuration is appropriate for the target machine; however, these systems cannot separate custom configuration variables from erroneous ones since they do not observe how applications actually use those values. In contrast, ConfAid can differentiate these cases by observing how the values are used inside the application bi-

nary. Unlike X-ray, PeerPressure and Strider are not suitable for diagnosing performance problems, because they analyze the static state rather than observe applications execute.

Chronus [73] also compares multiple configuration states. Instead of comparing states across computers, it uses virtual machine checkpoint and rollback to "time travel" through states on the same machine, looking for the instance in which the program behavior on a particular test case switched from correct to incorrect. Unlike Chronus, the tools introduced in this thesis do not require a prior state where the application worked correctly. Therefore, we can troubleshoot configuration of new applications and new features in already-existing applications.

AutoBash [66] uses causality analysis inside the OS kernel to improve misconfiguration troubleshooting. SigConf improves the black-box approach of AutoBash by capturing more information about predicate executions than a simple success/failure state. ConfAid can identify finer-grained root causes compared to AutoBash, but unlike AutoBash, ConfAid only focuses on root causes in configuration files. AutoBash did not handle misconfigurations that lead to performance problems.

Another approach to improve configuration management is to proactively detect situations that may lead to configuration errors in the future. For example, CODE [82] is a tool that uses machine learning algorithms to learn correct sequences of Windows registry accesses and raise a warning when an access violates a correct sequence. Similar to other machine learning approaches, CODE needs to observe a sequence several times before it can classify that sequence. Therefore, CODE may falsely flag rare but correct accesses as wrong accesses, and it cannot judge new sequences.

As another example of proactive detection, Barricade [53] is a system that tries to detect and confine mistakes in large systems. Barricade employs a combination of testing, error detection, cost analysis, and confinement to achieve this goal. However, Barricade only works for frequently performed configuration tasks, and it heavily relies on expert users to provide task descriptions and test units. In general, the approach of systems like CODE and

Barricade is orthogonal to our solutions, and they can be combined to provide a stronger configuration management system.

The most common way to troubleshoot software problems is to analyze log messages and error reports generated by the application. Unfortunately, error and log messages are usually too cryptic to lead the user to the root cause of the problem. LogEnhancer [83] enhances the application log messages by adding debugging data, such as values of relevant variables. The target audience of this tool is developers, who can benefit from low-level, code-related debugging information. Clarify [33] improves error reporting by generating signatures using program features such as function call counts, call sites, and stack dumps. It then classifies the signatures using machine learning techniques. While these tools improve the quality of log messages, the user still needs to manually infer the root cause of the problem. The goal of our tools is to close this circle for configuration root causes.

Xu et. al. [75] and Lou et. al. [43] used machine learning techniques to analyze console log messages to detect problems in large systems. Our tools currently rely on the user to detect a problem. We can combine our tools with these detection tools to provide both detection and diagnosis.

Many other systems trace causality for purposes other than troubleshooting. For example, taint tracking [52] monitors data flow dependencies to determine when input data is used in an insecure manner. RedFlag [24] uses data flow analysis to reduce the leaks of sensitive information by personal machines. Resin [78] uses application-level data flow assertions to improve the security of applications. Decentralized information flow [49, 85] monitors both control flow and data flow dependencies to determine if a code component leaks information that it is not authorized to divulge. PASS [47] uses causality to annotate files with provenance that describes their causal inputs. BackTracker [41] traces causal interactions to determine what state has been changed during an intrusion. Asbestos [28] and HiStar [86] monitor causality in the OS to prevent inadvertent disclosure of private data.

Symbolic execution is another type of causality analysis, where the inputs of the system

are propagated in an abstract form to enable exploration of more paths in the application code [12, 19, 76]. While our tools leverage the general idea of causality analysis, the focus of symbolic execution systems is usually very different from our tools. Our tools start from a specific undesired execution and try to explain the reason it happened; whereas symbolic execution tools usually start from an abstract input and explore multiple paths to infer the impacts of different concrete inputs.

The next three sections discuss prior works that are closely related to each one of our diagnosis tools.

6.1 SigConf

Similar to SigConf, Chronus 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. SigConf, however, cannot diagnose bugs that do not exist in the reference computer database.

Su et. al. [67] propose a system that automatically generates predicates by observing human actions trying to solve a configuration problem. Such systems can be leveraged to generate predicates that are later used by SigConf for diagnosing a configuration problem.

Similar to our method, Yuan *et al.* [80] 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 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.

Bodik et. al. [8] use statistical metrics to generate signatures for the performance of

datacenters. These signatures are compared against signatures of previously solved cases to quickly troubleshoot a known performance problem. The work of Bodik et. al. and SigConf both use the approach of troubleshooting via signature comparison. However, SigConf uses causality analysis to generate signatures that capture the interactions between the application and the operating system; whereas the tool introduced by Bodik et. al. captures the performance characteristics of systems.

6.2 ConfAid

Dytan [21] proposes a generic dynamic taint analysis framework to ease the implementation of various taint-based techniques. ConfAid enhances the basic dynamic taint analysis with essential heuristics and applies it to configuration troubleshooting problem.

Some systems leverage white box analysis to help developers replicate a problem experienced in the field. SherLog [81], ESD [84], and the work of Crameri et. al. [25] use static analysis and symbolic execution to infer the execution path of the application. SherLog uses log messages, and ESD and Crameri et. al. leverage the bug report generated by the application to constrain the execution path. These systems can replicate an execution path that derives from a misconfiguration. However, they make different design decisions than ConfAid, driven by their different use case. They require application source code, and SherLog also may require guidance from developers about which functions should be symbolically executed. This is appropriate for a tool used by software experts, but less so for one like ConfAid that is targeted at administrators and users.

Program slicing [1, 88, 87], intended to aid in debugging, is a more general approach that determines which statements could affect the value of a variable using a backward or forward computations. ConfAid applies similar data and control flow analysis techniques to a new problem, namely determining the root causes of misconfigurations.

ConfAid uses deterministic record and replay. While many prior software systems provide this functionality [3, 27, 31, 56, 64, 69, 64], ConfAid introduces new constraints that

prior systems cannot satisfy. The fidelity of replay must be high enough to exactly reproduce application instructions and system calls, while still being loose enough to execution instrumentation during the replayed execution but not during the recorded execution. ConfAid modified the replay system to compensate for the instrumentation code in order to achieve the needed fidelity.

Aftersight [20] also decouples program execution from analysis. However, it performs the record and replay tasks, as well as the analysis task, in the VMM layer. Our deterministic record and replay system is specifically designed to work with Pin, since both ConfAid and X-ray use Pin to perform their analysis.

6.3 X-ray

Profilers such as OProfile [55], VTune [70], Fay [29], DTrace [13], SystemTap [59], ETW [46], Debox [61], and Chopstix [7] allow the troubleshooter to instrument applications and/or the operating system and collect performance data. These tools reveal *what* events (e.g., functions) incur substantial performance costs. However, their users must manually infer *why* those events executed. In contrast, X-ray automatically identifies root causes.

Other tracing systems follow activities across multiple components or protocol layers, and use the causal relationships they observe to propagate and merge performance data. X-trace [30] observes network activities across protocols and layers in a distributed system. SNAP [79] profiles TCP-statistics and socket-call logs and correlates data across a data center. Aguilera *et al*. [2] use statistical analysis to infer causal paths between application components and attribute delays to specific nodes. Pinpoint [18] traces communication between middleware components to infer which components are responsible for causing faults. Follow-on work [17] adds the abstraction of causal *paths* that link black-box components. Like these tools, X-ray uses causality to propagate data across components when processes communicate (although propagation is currently limited to a single node by its

replay system). Unlike these tools, X-ray analyzes causality *within* application components using dynamic binary instrumentation, so it can determine the specific relationship between component inputs and outputs.

Other performance troubleshooting tools build or use a model of application performance. Magpie [5] accurately extracts the control flow and resource consumption of each request to build a workload model that can be used for performance prediction. Magpie's per-request profiling can help troubleshooters diagnose potential performance problems. Even though Magpie provides detailed performance information that can be used to manually infer root causes, it still does not automatically diagnose *why* the observed performance anomalies occur. Magpie uses schemas to determine which requests are being executed by various components; X-ray currently uses a simpler method and thus could benefit from using Magpie's schemas for complicated request patterns.

Stewart *et al.* [65] extract resource usage from multi-component services to generate performance models for capacity planning and cost-effectiveness analysis. Urgaonkar *et al.* [4] use resource usage profiling to guide application placement in shared hosting platforms. Cohen *et al.* [22] use statistical learning techniques to automatically build system models. They identify a combination of system-level metrics and threshold values that correlate with high-level performance states. In contrast to X-ray, none of these systems tie performance to specific root causes such as configuration options.

Many research projects tune performance [26, 16, 89] by injecting artificial traffic and using machine learning to correlate performance with specific configuration options. Unlike X-ray, these tools limit the set of configuration options analyzed, and they must see controlled traffic in order to learn good configuration values.

Spectroscope [62] diagnoses performance changes by comparing request flows between two executions of the same workload. Kasick *et al.* [39] compare similar requests to diagnose performance bugs in parallel file systems. Unlike X-ray, these tools must see very similar requests in order to diagnose performance problems. In contrast, our results show

that X-ray can correctly identify root causes even when requests are very dissimilar because it analyzes the control path of each request.

As mentioned in section 6.1, Bodik et. al. [8] use statistical metrics to diagnose performance problems that have happened before in datacenters. Unlike this tool, X-ray can be used to troubleshoot problems that have not previously happened. X-ray only considers root causes from configuration files and user input, but this tool can diagnose known problems with other root causes.

CHAPTER VII

Conclusion

This chapter describes our plans to extend our work in configuration troubleshooting and summarizes the contributions of this thesis.

7.1 Future directions

In this thesis, we focused on diagnosing misconfiguration problems. Before the diagnosis begins, we rely on the end user or the administrator to detect the problem. Then, we automatically diagnose the root cause of the problem specified by the user, and report the diagnosis results. The user then needs to determine what actions must be taken to actually fix that problem. As future research directions, we would like to also automate the detection and fixing tasks.

7.1.1 Detecting anomalies

Detecting anomalies is challenging in complex systems. This problem is exacerbated for production environments. The reason is that production software usually does not collect much diagnosis information to maximize performance. With little debugging information available, detecting anomalies becomes very challenging for these systems. We would like to explore the possibility of overcoming this difficult tradeoff by automatically detecting anomalies online with low overhead.

Anomalies may manifest as failures, crashes, incorrect behavior, or simply poor performance. We plan to start by considering anomalies in which the application execution path is different from a normal run, e.g. a problem that causes the application to perform extra network activities, which result in an abnormally high latency. Thus, executing an uncommon path can be an indication of an anomalous behavior. The problem of anomaly detection is well-studied in the security community. We have successfully borrowed techniques from the security community in the past for the ConfAid project. Our initial idea for solving this problem is also to investigate whether these anomaly detection techniques can be re-purposed for our problem, and can be altered to incur less overhead.

7.1.2 Fixing configuration problems

Diagnosing the root cause of a misconfiguration usually simplifies the fixing process. However, automatically determining correct actions that solve a misconfiguration and do not cause other problems is still challenging. As a future research direction, we plan to further explore this problem.

We plan to tackle this challenge by first considering misconfiguration problems that can be solved by modifying the values of configuration parameters in configuration files. We first use a tool like ConfAid to determine which configuration parameters are most likely the root causes of the problem. Proposing a correct value for a misconfigured parameter raises two challenges. First, we need to automatically and efficiently find potential values for the parameter. While this issue is trivial for binary values, e.g. *yes* or *no* parameters, it is quite difficult for values that involve paths or numbers. We plan to explore techniques such as symbolic execution to narrow down the possibilities.

Once we propose a new value for a parameter, we need to determine whether the modification actually solves the problem. Thus, the second challenge is to automatically determine whether an execution is resulting in a failure or success. Some symptoms such as crashes or assertion failures are obvious signs of failing execution; however, many ex-

ecutions do not manifest such evident symptoms. We plan to leverage a history-based user-assisted approach to deduce the success or failure of an execution.

7.2 Contributions

This thesis demonstrates that we can automate misconfiguration diagnosis by analyzing the causal relationships between the inputs of an application and its output. We showed that these causal dependencies can be captured and analyzed at various granularities, without using the source code of the application. We built three misconfiguration diagnosis tools, SigConf, ConfAid, and X-ray, that leverage these causal relationships to pinpoint root causes of misconfiguration problems.

This thesis presents the details of design and implementation of these tools. In particular, we used coarse-grained causality analysis to create simple, cheap, and robust signatures that capture the state of a computer. SigConf uses these signatures to determine whether the current problem is similar to already known misconfigurations. We designed and implemented a fine-grained information flow analysis engine for x86 binaries that propagates information via data flow, control flow, and implicit control flow. ConfAid uses this engine to track configuration tokens as the application runs and link an incorrect output to the configuration parameters that caused it. We also introduced and implemented the idea of performance summarization and differential performance summarization in X-ray to diagnose performance misconfigurations. ConfAid and X-ray use our deterministic record and replay system to offload heavy-weight analysis from applications. Our replay system is instrumentation-aware, i.e., it allows the replayed execution to diverge from the recorded execution by running analysis code.

Our evaluation of SigConf, ConfAid, and X-ray on a variety of complex applications demonstrate that the idea of causality analysis can significantly improve misconfiguration diagnosis. We plan to provide the tools and infrastructures that we built to the wider research community.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 74–89, Bolton Landing, NY, October 2003.

[3] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 193–206, October 2009.

[4] P. S. B. Urgaonkar and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 239–254, Boston, MA, December 2002.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 259–272, San Francisco, CA, December 2004.

[6] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, May 2003.

[7] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, pages 103–116, San Diego, CA, December 2008.

[8] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Anderson. Fingerprinting the datacenter: Automated classification of performance crisis. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 111–124, 2010.

[9] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

[10] A. B. Brown and D. A. Patterson. To err is human. In *DSN Workshop on Evaluating and Architecting System Dependability*, Goteborg, Sweden, July 2001.

[11] A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the 2003 USENIX Technical Conference*, San Antonio, TX, June 2003.

[12] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Usenix Symposium on Operating System Design and Implementation (OSDI)*, pages 209–224, December 2008.

[13] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 15–28, Boston, MA, June 2004.

[14] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[15] A. Chanda, K. Elmeleegy, A. L. Cox, and W. Zwaenepoel. Causeway: Operating system support for controlling and analyzing the execution of distributed programs. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)*, Santa Fe, NM, June 2005.

[16] H. Chen, G. Jiang, H. Zhang, and K. Yoshihira. Boosting the performance of computing systems through adaptive configuration tuning. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1045–1049, Honolulu, Hawaii, March 2009.

[17] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

[18] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic Internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 595–604, Bethesda, MD, June 2002.

[19] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in vivo multi-path analysis of software systems. In *ASPLOS*, March 2011.

[20] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Technical Conference*, pages 1–14, June 2008.

[21] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *In Proceedings of the International Symposium on Software Testing and Analysis*, pages 196–206, July 2007.

[22] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 231–244, San Francisco, CA, December 2004.

[23] Computing Research Association. Final report of the CRA conference on grand research challenges in information systems. Technical report, September 2003.

[24] L. P. Cox and P. Gilbert. RedFlag: Reducing inadvertant leaks by personal machines. Technical Report MSR-TR-2009-02, Duke University, 2009.

[25] O. Crameri, R. Bianchini, and W. Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *Proceedings of the 6th European Conference on Computer Systems*, EuroSys '11, pages 199–214, 2011.

[26] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus. Managing Web Server Performance with AutoTune Agent. *IBM Systems Journal*, 42(1):136–149, January 2003.

[27] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Boston, MA, December 2002.

[28] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, United Kingdom, October 2005.

[29] U. Erlingsson, M. Peinado, S. Peter, and M. Budiu. Fay: Extensible distributed tracing from kernels to clusters. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 311–326, October 2011.

[30] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th NSDI*, pages 271–284, Cambridge, MA, April 2007.

[31] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proceedings of the USENIX 2006 Annual Technical Conference*, Boston, MA, June 2006.

[32] J. Gray. Why do computer stop and what can be done about it? Technical Report 85.7, Tandem Corp., June 1985.

[33] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black-box components. In *Proceedings of the Conference on Programming Language Design and Implementation 2007*, pages 101–111, San Diego, CA, 2007.

[34] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

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

[36] R. Johnson. More details on today's outage. http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919.

[37] F. Junqueira, Y. J. Song, and B. Reed. BFT for the skeptics. In *ACM Symposium on Operating Systems Principles: Work in Progress Session*, October 2009.

[38] A. Kapoor. Web-to-host: Reducing total cost of ownership. Technical Report 200503, The Tolly Group, May 2000.

[39] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-box problem diagnosis in parallel file systems. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, San Jose, CA, February 2010.

[40] 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)*, pages 157–166, Anchorage, AK, June 2008.

[41] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 223–236, Bolton Landing, NY, October 2003.

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

[43] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining invariants from console logs for system problem detection. In *Proceedings of the USENIX Annual Technical Conference*, pages 24–24, Boston, MA, June 2010.

[44] 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 *Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[45] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[46] http://msdn.microsoft.com/en-us/library/bb968803(v=VS.85).aspx.

[47] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 43–56, Boston, MA, May/June 2006.

[48] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International*, 11(5), 1995.

[49] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the Annual Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, TX, January 1999.

[50] K. Nagaraja, F. Oliveria, R. Bianchini, R. P. Martin, and T. Nguyen. Understanding and dealing with operator mistakes in Internet services. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 61–76, San Francisco, CA, December 2004.

[51] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007.

[52] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *In Proceedings of the 12th Network and Distributed Systems Security Symposium*, February 2005.

[53] F. Oliveria, A. Tjang, R. Bianchini, R. P. Martin, and T. D. Nguyen. Barricade: Defending systems against operator mistakes. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 83–96, 2010.

[54] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.

[55] http://oprofile.sourceforge.net/.

[56] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd SOSP*, pages 177–191, October 2009.

[57] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A framework for determinisrtic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, March 2010.

[58] http://www.karoltomala.com/blog/?p=576.

[59] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the Linux Symposium*, pages 49–64, Ottawa, ON, Canada, July 2005.

[60] F. Qian, K. S. Quah, J. Huang, J. Erman, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Web caching on smartphones: Ideal vs. reality. In *Proceedings of the 10th International Conference on Mobile Systems, Applications and Services*, Low Wood Bay, United Kingdom, June 2012.

[61] Y. Ruan and V. Pai. Making the "box" transparent: System call performance as a first-class result. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Boston, MA, June 2004.

[62] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th NSDI*, pages 43–56, Boston, MA, March 2011.

[63] Circleid, misconfiguration brings down entire .se domain in sweden. http://www.circleid.com/posts/misconfiguration_brings_down_entire_se_domain_-in_sweden.

[64] S. Srinivasan, C. Andrews, S. Kandula, and Y. Zhou. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Technical Conference*, pages 29–44, Boston, MA, June 2004.

[65] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *Proceedings of the Second Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[66] 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*, pages 237–250, Stevenson, WA, October 2007.

[67] Y.-Y. Su and J. Flinn. Automatically generating predicates and solutions for configuration troubleshooting. In *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, June 2009.

[68] K. Veeraraghavan, J. Flinn, E. B. Nightingale, and B. Noble. quFiles: The right file at the right time. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, pages 1–14, San Jose, CA, February 2010.

[69] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Long Beach, CA, March 2011.

[70] http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[71] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th Symposium on*

Operating Systems Design and Implementation, pages 245–257, San Francisco, CA, December 2004.

[72] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. STRIDER: A black-box, state-based approach to change and configuration management and support. In *Proceedings of the USENIX Large Installation Systems Administration Conference*, pages 159–172, October 2003.

[73] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 77–90, San Francisco, CA, December 2004.

[74] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 2007 Workshop on Modeling, Benchmarking and Simulation (MoBS)*, June 2007.

[75] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordani. Detecting large-scale system problems by mining console logs. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, October 2009.

[76] J. Yang, C. Sar, and D. Engler. eXplode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 131–146, Seattle, WA, November 2006.

[77] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. Bairavasundaram, and S. Pasupathy. An emprirical study on configuration errors in commerical and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, October 2011.

[78] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 291–304, October 2009.

[79] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *Proceedings of the 8th NSDI*, pages 57–70, Boston, MA, March 2011.

[80] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *Proceedings of the European Conference on Computer Systems*, Leuven, Belgium, 2006.

[81] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–154, Pittsburgh, PA, March 2010.

[82] D. Yuan, Y. Xie, R. Panigrahi, J. Yang, C. Verbowski, and A. Kumar. Context-based online configuration error detection. In *Proceedings of the USENIX Annual Technical Conference*, pages 28–28, Portland, OR, June 2011.

[83] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–14, NewportBeach, CA, March 2011.

[84] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the European Conference on Computer Systems*, pages 321–334, April 2010.

[85] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 1–14, Banff, Canada, October 2001.

[86] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieres. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

[87] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 169–180, June 2006.

[88] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards locating execution omission errors. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 415–424, June 2007.

[89] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic configuration of Internet services. In *Proceedings of the European Conference on Computer Systems*, pages 219–229, Lisbon, Portugal, March 2007.