Automating configuration troubleshooting with dynamic information flow analysis

Mona Attariyan and Jason Flinn
University of Michigan

Abstract

Software misconfigurations are time-consuming and enormously frustrating to troubleshoot. In this paper, we show that dynamic information flow analysis helps solve these problems by pinpointing the root cause of configuration errors. We have built a tool called ConfAid that instruments application binaries to monitor the causal dependencies introduced through control and data flow as the program executes — ConfAid uses these dependencies to link the erroneous behavior to specific tokens in configuration files. Our results using ConfAid to solve misconfigurations in OpenSSH, Apache, and Postfix show that ConfAid identifies the source of the misconfiguration as the first or second most likely root cause for 18 out of 18 real-world configuration errors and for 55 out of 60 randomly generated errors. ConfAid runs in only a few minutes, making it an attractive alternative to manual debugging.

1 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. The cost of troubleshooting is substantial. Technical support contributes 17% of the total cost of ownership of today's desktop computers [24], 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 [16]. Even for casual computer users, troubleshooting is often enormously frustrating.

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

Why address misconfigurations specifically? Empirical evidence exists that misconfigurations are often the dominant cause of problems in deployed systems. For example, Gray [20] attributed 42% of system outages to administration, while software, hardware, and environment failures account for 25%, 18%, and 14% of failures, respectively. Murphy and Gent [31] 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. Other studies have shown that configuration errors are the largest category of operator mistakes. Oppenheimer et al. [35] studied three commercial Internet services and found that more than 50% of the operator mistakes that led to service unavailability were misconfigurations. Nagaraja et al. [33] found that software misconfiguration was the most common type of operator mistake, accounting for more than half of all mistakes. Other studies have shown similar results [7, 8, 23]. Further, while fault tolerance techniques such as modular redundancy [30] or Byzantine fault tolerance [10] can mask software and hardware faults, they do not prevent human error such as an operator who misconfigures all replicas [20, 23].

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

tially increase the time to recover (TTR) from a failure.

To remedy this problem, we have developed a tool, called ConfAid, that uses dynamic information flow analysis to identify the likely root cause of a configuration problem. When a user or administrator wishes 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. 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, 11, 12, 13], 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 [29], 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.

2 Design principles

We next briefly describe ConfAid's design principles.

2.1 Use white-box analysis

The genesis of ConfAid arose from AutoBash [37], our prior work in configuration troubleshooting. AutoBash tracks causality at process and file granularity in order to diagnose configuration errors. It treats each process as a *black box*, such that all outputs of the process are considered to be dependent on all prior inputs. We found AutoBash to be very successful in identifying the root cause of problems, but the success was limited in that AutoBash would often identify a complex configuration file, such as Apache's `httpd.conf`, as the source of an error. When such files contain hundreds of options, the root cause identification of the entire file is often too nebulous to be of great use.

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

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.

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 [29] to dynamically insert instrumentation into binaries as applications run. It also uses IDA Pro [22] to statically generate control flow graphs from binaries.

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 Design and implementation

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. For non-deterministic bugs, ConfAid could potentially leverage one of several deterministic replay systems that can capture a buggy non-deterministic execution and faithfully reproduce it for later analysis [3, 18, 27, 36].

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 3.5. 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 TTR.

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 and 3.4. We also assume that ConfAid is tracking only a single process; Section 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 [29] 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 [34] to analyze information flow. It inserts instrumentation into the binary that 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

```
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 1: Example to illustrate causality tracking

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 3.4, we attach a weight to each taint.

Taint is propagated via data flow and control flow dependencies. When a monitored process executes an instruction that modifies a memory address, register, or CPU flag, the taint set of each modified location is set to the union of the taint sets of the values read by the instruction. For example, 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 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 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. [48] 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, applica-

tion 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 3.3.1 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 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), 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

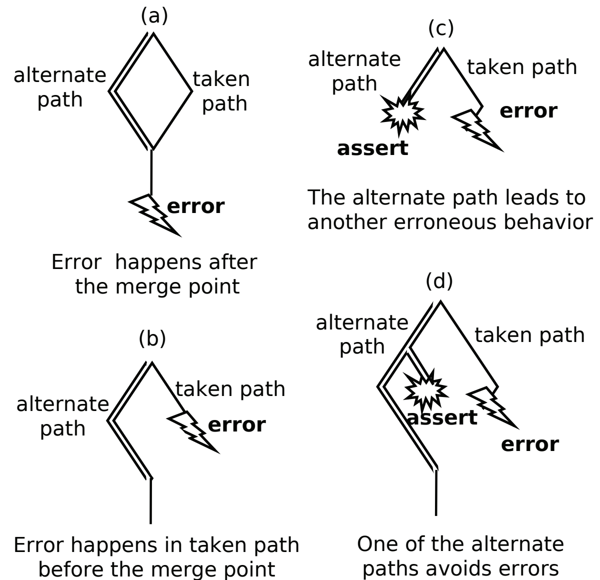

Figure 2: Examples illustrating ConfAid path analysis

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.

3.2.1 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 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.

3.3.1 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.

3.3.2 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 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. 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.

```
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: Example to illustrate alternate path pruning

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 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, $T_{alt} = T_{c1} \cap T_{c2}$ 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, 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.

3.4 Heuristics for reducing false positives

We originally designed ConfAid to use only the basic taint tracking algorithm described in Section 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 floating-point 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 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 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

```
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 4: Example to illustrate the weighting heuristic

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, $T_y = max(T_y, min(\frac{T_{c1}}{4}, \frac{T_{c2}}{2}))$.

Figure 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 sce-

nario is uncommon, we did observe a single Apache bug (described further in the evaluation) in which ConfAid's heuristic failed in this manner.

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.

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

Second, like previous configuration troubleshooting systems [38, 39], 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 3.3.2. 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.

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?

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

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 1 lists the configuration errors for each application, as well as the ConfAid results.

In these experiments, 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 varies depending 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 that application starts only a single process.

As shown in Table 1, 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 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 1 is taken from a thread in linuxforums.org [28]. 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.

4.3 Effect of the weighting heuristic

We next examine the effect of the weighting heuristic introduced in Section 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 1 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.

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 3.3.1) from the default value of 80 instructions. As Figure 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

Application	Bug	Description of misconfiguration	Total # of options	ConfAid rank of the root cause	Execution time	# false positives w/o weights
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.	47	2^{nd}(tied w/1)	1m 16s	6
	2	The server only has the PasswordAuthentication option enabled, while the user can only authenticate via RSA keys.	47	1^{st}(tied w/1)	1m 10s	1
	3	The user does not have his public key in the directory specified in the SSH server config file. Therefore, he cannot authenticate.	48	2nd	51s	43
	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.	49	2nd	48s	44
	5	The MaxAuthTries option in SSH server config is set too low. Therefore, the user is disconnected if she enters her password incorrectly once.	47	1st	1m 13s	43
	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.	47	1st	9s	0
	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.	47	1^{st}(tied w/1)	36s	43
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.	88	2^{nd}(tied w/1)	2m 46s	87
	2	The cgi-bin directory is ScriptAlised in the config file. This prevents the DirectoryIndex from working as expected. Therefore, the user cannot access the index file in the directory.	89	1st	2m 45s	87
	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.	89	2^{nd}(tied w/1)	2m 45s	88
	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.	93	1st	2m 59s	91
	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.	89	1st	2m 46s	88
	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.	89	1^{st}(tied w/1)	2m 47s	86
Postfix	1	The mydestination option is not set correctly in the Postfix config file. Therefore, Postfix cannot deliver mail locally.	27	1st	37s	4
	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.	27	1st	1m 10s	4
	3	The relayhost option is set incorrectly. Therefore, Postfix cannot forward the email sent from the user's machine to the Internet.	29	1st	47s	4
	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.	29	1st	32s	2
	5	The email address provided in luser-replay is not reachable. Therefore, Postfix cannot redirect other mail with wrong recipient to the luser-replay.	29	1st	1m 38s	0

Table 1: Results for 18 real-world configuration bugs

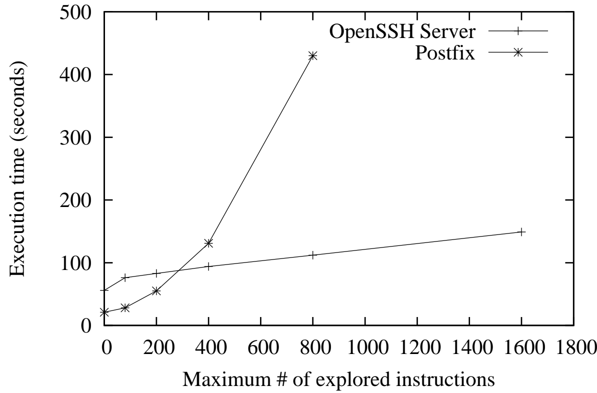

Figure 5: The effect of varying the horizon

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. 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.

4.5 Random fault injection

We next used ConfErr [25] 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 2, 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 success 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 3.4 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.

5 Related work

Several prior research efforts have applied different techniques to the problem of configuration troubleshooting. Unlike ConfAid, most prior systems have taken a *black box* approach that uses only state external to the application being debugged to infer the problem.

PeerPressure [38] and its predecessor, Strider [39], use statistical methods to compare configuration state in the Windows registry on different machines. When a value

Application	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	Avg. time to run
OpenSSH	17 (85%)	1 (5%)	1 (5%)	0	1 (5%)	7s
Apache	17 (85%)	1 (5%)	0	1 (5%)	1 (5%)	24s
Postfix	15 (75%)	0	2 (10%)	0	3 (15%)	38s

Table 2: Random fault injection results

on a machine exhibiting erroneous behavior differs from the value usually chosen by other machines, PeerPressure flags the value as a potential error. This approach works well as long as the majority configuration is appropriate for the target machine; however, PeerPressure and Strider 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 binary.

Chronus [40] 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.

Other projects monitor state external to applications as they run. Cohen et al. [15] use statistical techniques to help troubleshoot performance issues by correlating those issues with low-level performance statistics for the CPU, disk, and other system components. AutoBash [37] traces causality inside the OS by monitoring system call execution, but treats execution inside each process as a black box. AutoBash can suggest that a particular configuration file may be erroneous, but it cannot identify the specific value within the file that is at fault.

Our previous work on misconfiguration diagnosis [4] uses the application's system call trace to extract the files and processes on which the application causally depends. It then generates a signature based on those dependencies to represent the misconfiguration and search for the signature in a database of known bugs. Clarify [21] uses similar execution signatures to improve error reporting. It uses program features such as function call counts, call sites, and stack dumps to generate the signatures. The improved error reporting, although helpful, does not diagnose the root cause.

In contrast to all these projects, ConfAid takes a *white box* approach to configuration troubleshooting by monitoring causality within the program binary as it executes. Thus, ConfAid can observe the actual dependencies as they are introduced rather than inferring them through statistical and other methods.

Two recent systems apply white box analysis to a related problem: helping developers replicate a problem experienced in the field. SherLog [43] and ESD [44] both use static analysis and symbolic execution to infer the execution path of the application. SherLog uses log messages and ESD leverages the bug report generated by the application to constrain the execution path. Both of 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 both 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. More generally, symbolic execution systems have been applied to model checking file systems and other complex software systems [9, 41].

A number of systems trace causality external to processes to debug configuration and performance issues in distributed systems. For example, the work of Aguilera et al. [2] and Magpie [5, 6] trace RPCs and other network communication to debug performance problems. Causeway [11] allows applications to inject metadata that follows causal paths for distributed applications. Pinpoint [13] traces middleware and communications between components in a distributed system and statistically correlates traces with success and failure data. Follow-on work to Pinpoint [12] adds the abstraction of causal *paths* that link black-box components. ConfAid and these systems share the common idea of propagating causal information among distributed components; however, ConfAid also propagates causal information within processes, which allows it to precisely determine the causal relationships between inputs and outputs.

More generally, many systems reason about causal interactions in the operating system and in distributed systems. For example, taint tracking [34] monitors data flow dependencies to determine when input data is used in an insecure manner. ConfAid uses the same approach for data flow analysis, but applies it to a different domain. Dytan [14] 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. RedFlag [17] uses data flow analysis to reduce the leaks of

sensitive information by personal machines. Resin [42] uses application-level data flow assertions to improve the security of applications. Decentralized information flow [32, 45] monitors both control flow and data flow dependencies to determine if a code component leaks information that it is not authorized to divulge. BackTracker [26] traces causal interactions to determine what state has been changed during an intrusion. Asbestos [19] and HiStar [46] monitor causality in the OS to prevent inadvertent disclosure of private data.

Program slicing [1, 48, 47], 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.

6 Conclusion

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

Acknowledgments

We thank the anonymous reviewers and our shepherd, Shan Lu, for comments that improved this paper. We also thank the ConfErr team for helping us use their tool. This research was supported by NSF award CNS-1017148. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the University of Michigan, or the U.S. government.

References

[1] 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] M. Attariyan and J. Flinn. Using causality to diagnose configuration bugs. In *Proceedings of the USENIX Annual Technical Conference*, pages 171–177, Boston, MA, June 2008.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

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

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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.

[27] D. Lee, B. Wester, K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 77–89, Pittsburgh, PA, March 2010.

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

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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 ACM Symposium on Operating Systems Principles*, pages 177–191, October 2009.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 2010 European Conference on Computer Systems (EuroSys)*, pages 321–334, April 2010.

[45] 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.

[46] 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.

[47] 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.

[48] 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.