Which Configuration Option Should I Change?

Sai Zhang Michael D. Ernst
Department of Computer Science & Engineering
University of Washington, USA
{szhang, mernst}@cs.washington.edu

ABSTRACT

Modern software often exposes configuration options that enable users to customize its behavior. During software evolution, developers may change how the configuration options behave. When upgrading to a new software version, users may need to re-configure the software by changing the values of certain configuration options.

This paper addresses the following question during the evolution of a configurable software system: which configuration options should a user change to maintain the software's desired behavior? This paper presents a technique (and its tool implementation, called ConfSuggester) to troubleshoot configuration errors caused by software evolution. ConfSuggester uses dynamic profiling, execution trace comparison, and static analysis to link the undesired behavior to its root cause — a configuration option whose value can be changed to produce desired behavior from the new software version.

We evaluated ConfSuggester on 8 configuration errors from 6 configurable software systems written in Java. For 6 errors, the root-cause configuration option was ConfSuggester's first suggestion. For 1 error, the root cause was ConfSuggester's third suggestion. The root cause of the remaining error was ConfSuggester's sixth suggestion. Overall, ConfSuggester produced significantly better results than two existing techniques. ConfSuggester runs in just a few minutes, making it an attractive alternative to manual debugging.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging.
General Terms: Reliability, Experimentation.
Keywords: Configuration error diagnosis, Software evolution.

1. INTRODUCTION

Many modern software systems support a range of configuration options for users to customize their behavior. This flexibility has a cost: a small configuration error may cause hard-to-diagnose behavior.

Software configuration errors are errors in which the software code and the input are correct, but the an incorrect value is used for a configuration option so that the software does not behave as desired. Such errors may lead the software to crash, produce erroneous output, or simply perform poorly. In practice, software configuration errors are *prevalent*, *severe*, and *hard to debug*, but they are *actionable* for users to fix.

Prevalent. A recent analysis of Yahoo's mission-critical Zookeeper service showed that software misconfigurations accounted for the majority of all user-visible failures [52]. Configuration-related issues caused about 31% of all failures at a commercial storage company [69]. The vast majority of production failures at Google arise not due to bugs in the software, but bugs in the configuration settings (i.e., configuration errors) that control the software [15].

Severe. Configuration errors can have disastrous impacts. For example, an outage in Facebook due to an incorrect configuration value left the website inaccessible for about 2 hours [14]. The entire `.se` domain of Sweden was unavailable for about 1 hour, due to a DNS misconfiguration problem [49]. A misconfiguration made Microsoft's public cloud platform, Azure, unavailable for about two and a half hours [40]. Each such incident affected millions of users.

Hard to debug. Configuration errors are difficult to diagnose. They usually require great expertise to understand the error root causes. For example, a configuration error in the CentOS kernel prevented a user from mounting a newly-created file system [69]. The user needed deep understanding about the exhibited symptom, and had to re-install kernel modules and also modify configuration option values in several places to get it to work. Techniques to help escape from "configuration hell" are highly demanded [15].

Actionable. Unlike software bugs, which can only be fixed by experienced software developers, fixing a software configuration error is *actionable* for software end-users or system administrators. These users are not the software developers, and cannot access (much less understand) the source code; but they can fix a configuration error by simply changing the values of certain configuration options.

1.1 Configuration Evolution

Continual change is a fact of life for software systems. Among software changes, configuration changes are prevalent. We studied 8 real-world configurable software systems (Section 2), and found configuration changes in *every* studied version of *each* system. In many cases, reusing the old version's configuration can lead the new software version to exhibit *undesired* behaviors, even if the software is working exactly as *designed*.

Take the popular JMeter performance testing tool as an example. In version 2.8, the testing report is saved as an XML file after running an example command (`jmeter -n -t ../threadgroup.jmx -l ../output.jtl -j ../test.log`) from the user manual. However, after upgrading to version 2.9, the same command saves the testing report in a CSV file. Further, all JMeter regression tests pass on the updated version. The new JMeter version behaves as *designed* but *differently* than a user was expecting.

Our technique (and its tool implementation ConfSuggester) can help diagnose configuration errors. For the JMeter example, a user

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05...$15.00
http://dx.doi.org/10.1145/2568225.2568251

first demonstrates the different behaviors on two ConfSuggester-instrumented JMeter versions. Then, ConfSuggester analyzes the recorded execution traces produced by the two instrumented versions, and outputs a ranked list of suspicious configuration options that may need to be changed. At the top of the list is the `output_format` option with a default value of `CSV` in version 2.9. To resolve this problem, users only need to change its value to `XML`.

1.2 Configuration Option Recommendation

Broadly speaking, diagnosing a configuration error can be divided into three separate tasks: reproducing the error, recommending which specific configuration option is responsible for the undesired behavior, and determining a better value for the configuration option to fix the error. ConfSuggester addresses the second task: recommending the root-cause configuration option.

ConfSuggester specifically focuses on software configuration errors and aims to help two types of users: software end-users who may have problems with software installed on their personal computers, and system administrators who are responsible for maintaining production systems. They can use ConfSuggester to diagnose an unexpected configuration problem during software evolution.

The key idea of ConfSuggester is to approximate program behavioral differences by control flow differences between two executions (by running the old and new program versions, respectively), and then reason about the control flow differences to identify configuration options that might cause such differences. It uses three steps, as illustrated in Figure 5, to link the undesired behavior to specific root-cause configuration options:

- **Instrumentation and Profiling.** ConfSuggester instruments both the old and new program versions to monitor the execution of each statement as well as the evaluation result of every predicate. Then, it asks the user to demonstrate the different behaviors on the two instrumented program versions.
- **Execution Trace Comparison.** ConfSuggester analyzes the two execution traces to identify the control flow differences. ConfSuggester identifies program predicates that behave differently between the two versions. These behaviorally-deviated predicates and their affected program statements provide evidence about which parts of a program might be behaving abnormally and why.
- **Configuration Option Recommendation.** ConfSuggester uses a lightweight static dependence analysis technique, called thin slicing [53], to attribute control flow differences to specific configuration options. Finally, it outputs a ranked list of suspicious options to the users.

Compared to existing error diagnosis techniques [4–6, 47, 55, 63, 66, 74], ConfSuggester differs in four key aspects:

- **It diagnoses configuration errors caused by software evolution**. Most existing configuration error diagnosis techniques identify errors from a single program version [4–6, 47, 55, 63, 66, 74]. By contrast, ConfSuggester is cognizant of software evolution and works on two different versions of the same program. It uses the desired behavior of the old software version as a baseline against which to compare new program behavior, and only reasons about the behavioral differences.
- **It requires no testing oracle**. Some previous work [6, 47, 55, 66] requires the user to answer difficult questions like "is the software currently working?" or "why is the software not working?" by writing a testing oracle to check the software behavior. By contrast, ConfSuggester only requires users to demonstrate the different behaviors on two versions. ConfSuggester uses the execution trace produced by the old version as an approximate oracle to reason about the undesired behavior on the new version.
- **It determines likely root-cause options**. Many error diagnosis and debugging techniques [71, 75] primarily focus on determining *what* causes the undesired behaviors, e.g., a snippet of code — they leave the more challenging question of *how* to fix the undesired behaviors unanswered. The user must manually inspect the analysis report to infer the root cause, e.g., a configuration option. By contrast, ConfSuggester makes reports in terms that end-users can act on: it explicitly guides users to specific configuration options that may fix the error.
- **It requires no OS-level support**. ConfSuggester does not need alterations to the JVM, operating system, or standard library. This makes ConfSuggester more portable and distinguishes it from related techniques, such as OS-level configuration error diagnosis [55, 66].

1.3 Evaluation

We implemented ConfSuggester for Java software and empirically evaluated its effectiveness using 8 configuration errors from 6 open-source configurable Java software systems. We used ConfSuggester to recommend configuration options whose values can be changed to fix each error. ConfSuggester successfully recommended correct configuration options for all 8 errors. For 6 errors, the correct option was ConfSuggester's first suggestion. For 1 error, the correct option was ConfSuggester's third suggestion. The root cause of the remaining error was ConfSuggester's sixth suggestion. ConfSuggester is fast enough for practical use, taking less than 3.1 minutes, on average, to diagnose each configuration error. ConfSuggester's accuracy and speed make it a promising technique.

We compared ConfSuggester to two existing configuration error diagnosis techniques, called ConfDiagnoser [74] and ConfAnalyzer [47]. ConfDiagnoser assumes the existence of some correct execution traces on the new program version; by contrast, ConfSuggester eliminates the assumption. ConfAnalyzer exclusively focuses on diagnosing crashing configuration errors; by contrast, ConfSuggester can diagnose both crashing errors and non-crashing errors. Our experiments show that ConfSuggester significantly outperforms these two existing techniques.

Finally, we evaluated two internal design choices of ConfSuggester. First, we showed that using thin slicing [53] is a better choice than traditional full slicing [22] to reason about root-cause configuration options. Second, we showed that ConfSuggester outperforms an alternative approach that solely uses predicate behavior change to reason about the root-cause configuration options.

1.4 Contributions

This paper makes the following main contributions:

- **Study of configuration changes.** We describe an empirical study of 8 configurable software systems. Our study indicates that configuration changes are frequent during software evolution (Section 2).
- **Technique.** We present a technique to diagnose configuration errors for evolving software. Our technique links undesired behaviors to specific responsible configuration options (Section 3).
- **Implementation.** We implemented our technique in a tool, called ConfSuggester, for Java software (Section 4). It is publicly available at: `http://config-errors.googlecode.com`.
- **Evaluation.** We applied ConfSuggester to 8 configuration errors from 6 configurable software systems, and compared it with existing techniques. The results show the accuracy and efficiency of ConfSuggester (Section 5).

Program	Versions	Years	LOC (latest version)	Language
MySQL	5.1, 5.5, 5.6, 5.7	3	1565212	C/C++
Apache	2.0, 2.2, 2.4	3	139178	C/C++
Firefox	7.0–22.0 (16 versions)	3	8237915	C/C++
Randoop	1.2.1, 1.3.2, 1.3.3	6	19511	Java
Weka	3.4, 3.5, 3.6, 3.7	2	288369	Java
JChord	1.0, 2.0, 2.1	4	26617	Java
Synoptic	0.04, 0.05, 0.1	2	19153	Java
JMeter	2.6, 2.7, 2.8, 2.9	2	91979	Java

Figure 1: The open-source software systems we studied and their characteristics. Column "Years" is the active development period for the selected versions.

Program	# Added Options	# Deleted Options	# Modified Options
MySQL	26	24	23
Apache	5	0	10
Firefox	28	7	56
Randoop	37	26	2
Weka	72	4	13
JChord	13	10	5
Synoptic	3	0	2
JMeter	17	3	12
Total	201	70	123

Figure 2: The total number of new, deleted, and modified configuration options for each subject program.

2. REAL-WORLD CONFIGURATION CHANGES

In the software engineering literature, despite a rich body of software change analysis work [11, 12, 35, 58, 70, 77], software configuration changes across multiple versions are less studied. Do configuration changes arise during software evolution in practice? This section describes an initial study of 8 real-world configurable systems to answer this question.

2.1 Subject Programs and Study Methodology

Figure 1 lists 8 open-source configurable systems used in our study. MySQL [41] is a popular relational database management system. Apache [1] has been the dominant HTTP server on the Internet since 1996. Firefox [16] is an open-source browser available on multiple platforms. Randoop [48] is an automated test generator for Java programs. Weka [65] is a toolkit that implements machine learning algorithms. JChord [25] is a program analysis platform that enables users to design and implement static and dynamic program analyses for Java. Synoptic [57] mines a finite state machine model representation of a system from logs. JMeter [27] is a tool to load-test functional behavior and measure performance. Each program is highly configurable, and has evolved over a considerable amount of time for the selected versions (2–6 years).

In our study, we manually examined the revision history of each subject program, and searched for 5 keywords ("configuration option", "add option", "delete option", "rename option", and "change option") in commit messages and in the change logs. We searched 7022 commit messages and 28 change log entries, in which 422 commit messages and 28 change log entries were matched. For each match, we read the description of the change and the "diff" of the original file to check whether a configuration option is changed. We collected 394 distinct configuration changes in total.

2.2 Findings

Figure 2 summarizes the identified configuration changes for each subject program. Figure 4 further classifies the configuration changes into four categories shown in Figure 3 (each change belongs to a single category)[1].

Change Type	**Description**
Bugs	Fix existing bugs
Renaming	Change the option name
Features	Add, remove, or modify features
Reliability	Improve reliability or performance

Figure 3: Types of configuration changes identified in our study from the subject programs in Figure 1.

Program	**# Changed Configuration Options**			
	Bugs	Renaming	Features	Reliability
MySQL	0	15	55	3
Apache	0	0	11	4
Firefox	28	0	55	8
Randoop	0	2	62	1
Weka	1	0	81	8
JChord	0	2	24	2
Synoptic	0	5	0	0
JMeter	7	0	18	7
Total	36	24	301	33

Figure 4: The number of configuration changes of each type.

As shown in Figure 2, configuration changes occur in the evolution of every subject program. In fact, they occur in every version of each subject program (not shown in Figure 2, due to space limits).

As shown in Figure 4, feature-related configuration changes are the largest group across all subject programs. These changes include adding new configuration options, deleting existing options, or modifying the default value of an option.

Configuration evolution can have unexpected impacts on program behavior. After configuration changes, reusing an old configuration may yield a misconfiguration, causing different results on the new version. Section 5 shows concrete examples.

2.3 Threats to Validity

Our findings apply in the context of our subject programs and methodology; they might not apply to arbitrary programs.

The configuration changes identified by our methodology are certainly not complete. Our keyword search might have missed some configuration changes. Our methodology only studies changes that are directly made to a software configuration option. We may miss code or environment changes that indirectly affect the software behavior and require users to re-configure the new software version.

3. TECHNIQUE

ConfSuggester models a configuration as a set of key-value pairs, where the keys are strings and the values have arbitrary type.

3.1 Overview

ConfSuggester is based on two key insights. First, a program's control flow, rather than data flow, often propagates the majority of the effects of a configuration option. In other words, a configuration option is mainly used as a "flag" that affects the program behavior by changing the runtime execution path. Second, the control flow differences between two execution traces approximate the behavioral differences of two versions; they provide evidence about which parts of the program are behaving abnormally and why.

Based on these two insights, ConfSuggester uses three steps to link different behaviors across program versions to specific con-

[1]The "Bugs" change type in Figure 3 represents that fixing some bugs led to changes in configurations rather than fixing some buggy configurations.

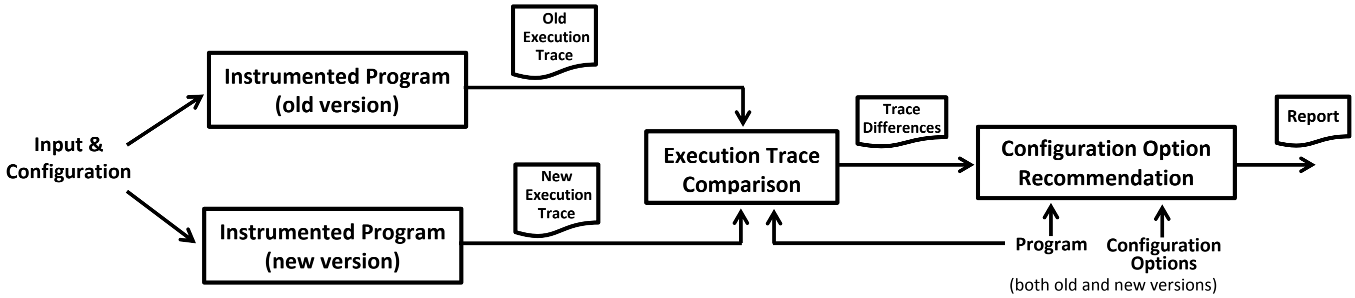

Figure 5: The architecture of our ConfSuggester technique. The instrumented program versions and two execution traces are produced by the step in Section 3.2. The "Execution Trace Comparison" step is described in Section 3.3. The "Configuration Option Recommendation" step is described in Section 3.4.

figuration options that cause the difference. Figure 5 sketches the high-level workflow of ConfSuggester. In the first step, ConfSuggester asks the user to demonstrate the different behaviors, using the same input and configuration, on two ConfSuggester-instrumented program versions (Section 3.2). Then, ConfSuggester identifies the control flow differences between the two execution traces produced by user demonstration. In particular, ConfSuggester identifies program predicates that behave differently across the two executions (Section 3.3). After that, ConfSuggester uses a lightweight dependence analysis technique, called thin slicing [53], to statically reason about which configuration options may cause the control flow differences. Finally, ConfSuggester reports a ranked list of suspicious configuration options to the user (Section 3.4).

3.2 Instrumentation and Demonstration

ConfSuggester first instruments both the old and new program versions to monitor the program execution at runtime. ConfSuggester directly instruments the bytecode. The instrumentation consists of two parts:

- For each program predicate (i.e., a branch instruction in bytecode), ConfSuggester inserts one probe before and one probe after it to monitor how frequently the predicate is executed and how often the predicate evaluates to true. In our context, a predicate is a Boolean expression in a conditional or loop statement, whose evaluation result affects the program control flow by determining whether to execute the following statement or not.
- For each of the other statements, ConfSuggester inserts one probe before it to monitor whether the statement gets executed or not at runtime. The statement execution information is used to calculate the number of executed statements controlled by a predicate (Section 3.4).

After instrumentation, ConfSuggester asks the user to demonstrate the different behaviors on the two instrumented program versions, using the same input and configuration. Demonstration is one of the simplest ways for an end-user to describe her problem; and it is easier than writing specifications or scripts of any form.

Executing the instrumented program produces an execution trace, which consists of a sequence of executed statements as well as the execution count and evaluation result of each predicate. The execution trace captured by ConfSuggester is by no means complete in recording the full program behavior; it only captures the control flows a program is taking. As demonstrated in our experiments, such control flow information serves as a good approximation to diagnose the undesired program behavior.

3.3 Execution Trace Comparison

In this step, ConfSuggester compares two execution traces from two program versions and identifies the control flow differences between them. ConfSuggester focuses on the recorded behavior of each predicate. First, it statically matches each predicate in the old source code to its counterpart in the new source code (Section 3.3.1). Then, it identifies all predicates that behave differently across the execution traces (Section 3.3.2).

3.3.1 Matching Predicates across Versions

For each predicate recorded in the old execution trace, ConfSuggester matches it in the new program version to identify its possibly-updated counterpart. The predicate-matching process proceeds in two steps. First, ConfSuggester finds corresponding methods. Then, ConfSuggester matches predicates within matched methods.

To match methods, ConfSuggester uses the first of these two strategies that succeeds:

1. **Identical method name.** Return a method with the identical fully-qualified name in the new version.
2. **Similar method content.** Return the method with the most similar content in the new version. Given a method in the old program version, ConfSuggester uses the algorithm shown in Figure 6 (details are discussed below) to match it to *every* method in the new program version, and then chooses the method in the new program version with the most matched statements.
   After running the matching algorithm, ConfSuggester further checks the ratio of matched statements in the old method, and discards method candidates whose matching ratio is below a threshold (default value: 0.9).

If there is no match for the declaring method in the new program version, ConfSuggester concludes that the predicate cannot be matched. Otherwise, ConfSuggester runs the algorithm in Figure 6 (or looks up a cached version of the result) to establish the mapping between instructions, and then returns the matched instruction of the predicate (or null if the predicate cannot be matched).

Statement-matching algorithm. The algorithm in Figure 6 is inspired by the JDiff program differencing algorithm [3]. The original JDiff algorithm is based on a method-level representation (called hammocks) that models object-oriented features. It works in a hierarchical way by first identifying matched classes and then matched method pairs, and uses textual similarity to compare two program statements. By contrast, our algorithm directly works on the bytecode, using the program control flow graph representation to establish the matching between statements.

In Figure 6, ConfSuggester first constructs the control flow graphs of two given methods (lines 2–3), then pushes their entry nodes (a synthetic node for each method) onto a worklist stack (line 5), which retains the next statement pair for comparison. The algorithm repeatedly pops a statement pair from the stack (line 7) and decides

Auxiliary functions:
matches(s, s'): return whether two statements s and s' are matched. Details are explained in Section 3.3.1.
BFS(s, cfg, d): return a list of statements reachable from statement s in cfg within d graph edges in Breath-First Search (BFS) order.
firstMatchedPair($stmtList_1$, $stmtList_2$): return the first matched statement pair $\langle s, s' \rangle$ such that $s \in stmtList_1$, $s' \in stmtList_2$, and matches(s, s') return true. Return null if no such pair exists.

Input: two methods from two software versions: m_{old} and m_{new}, a maximum lookahead value lh. (Our experiment uses $lh = 5$.)
Output: matched statements between m_{old} and m_{new}.

```
matchStatements(m_old, m_new, lh)
 1: matchedStmts ← new Map⟨Statement, Statement⟩
 2: cfg_old ← constructControlFlowGraph(m_old)
 3: cfg_new ← constructControlFlowGraph(m_new)
 4: stack ← new Stack⟨Pair⟨Statement, Statement⟩⟩
 5: stack.push(cfg_old.entry, cfg_new.entry)
 6: while stack is not empty do
 7:    ⟨stmt_old, stmt_new⟩ ← stack.pop()
 8:    if matchedStmts.keys().contains(stmt_old)
         || matchedStmts.values().contains(stmt_new) then
 9:       continue
10:    end if
11:    matchedPair ← null
12:    if matches(stmt_old, stmt_new) then
13:       matchedStmts[stmt_old] ← stmt_new
14:       matchedPair ← ⟨stmt_old, stmt_new⟩
15:    else
16:       stmtList_old ← BFS(stmt_old, cfg_old, lh)
17:       stmtList_new ← BFS(stmt_new, cfg_new, lh)
18:       ⟨s_old, s_new⟩ ← firstMatchedPair(stmtList_old, stmtList_new)
19:       if ⟨s_old, s_new⟩ ≠ null then
20:          matchedStmts[s_old] ← s_new
21:          matchedPair ← ⟨s_old, s_new⟩
22:       end if
23:    end if
24:    if matchedPair ≠ null then
25:       for each s in BFS(matchedPair.first(), cfg_old, 1) do
26:          for each s' in BFS(matchedPair.second(), cfg_new, 1) do
27:             stack.push(⟨s, s'⟩)
28:          end for
29:       end for
30:    end if
31: end while
32: return matchedStmts
```

Figure 6: Algorithm for matching statements from two methods.

whether the two statements are matched (line 12). Each statement appears at most once in the result (lines 8–9).

The algorithm decides whether two statements are matched by using the matches(s, s') auxiliary function. Method matches(s, s') returns true if both s and s' have the same statement type (i.e., the same instruction type in bytecode), and if s and s' are field-accessing or method-invoking statements, the same field or method is accessed or invoked by both statements. Such approximate matching tolerates small differences between two versions, such as changes to constant values.

If two statements are matched, the algorithm saves them in the result map (line 13). Otherwise, the algorithm compares each statement reachable within lh control flow graph edges (lines 16–22). Doing so permits the algorithm to tolerate some small changes in the method code, and attempts to match as many statements as possible.

When two matched statements are found (stored in the *matchedPair* variable in lines 14 or 21), the algorithm pushes every pair of their successor statements onto the stack (line 27). It terminates after every statement has been attempted to match.

3.3.2 *Identifying Behaviorally-Deviated Predicates*

Using the predicate matching information, ConfSuggester next identifies predicates that behave differently between two versions.

Given an execution trace T, ConfSuggester characterizes a predicate p's behavior by how often it is evaluated (i.e., the number of observed executions) and how often it evaluates to true (i.e., the "true ratio"). The true ratio is an important characteristic of a predicate's behavior, but it is less dependable the fewer times the predicate has been executed.

ConfSuggester combines the true ratio and number of executions by computing their harmonic mean.

$$\phi(p,T) = \frac{2}{\frac{1}{trueRatio(p,T)} + \frac{1}{totalExecNum(p,T)}}$$

In $\phi(p,T)$, $trueRatio(p,T)$ returns the proportion of executions of the predicate p that evaluated to true in T, and $totalExecNum(p,T)$ returns the the total number of observed executions of predicate p in T. To smooth corner cases, $\phi(p,T)$ returns 0, if a predicate p is not executed in T (i.e., $totalExecNum(p,T) = 0$) or a predicate p's true ratio is 0 (i.e., $trueRatio(p,T) = 0$). We let $\phi(null,T) = 0$ for all T.

Given two matched predicates p_1 and p_2 from two different execution traces T_1 and T_2, ConfSuggester uses the deviation function defined in Figure 7 to compute the behavioral deviation value. In Figure 7, the deviation function discards a predicate pair whose behavioral deviation value is less than a pre-defined threshold (line 2). This is for tolerating small non-determinism during program execution and making ConfSuggester focus on predicates with substantial behavioral differences.

The identified behaviorally-deviated predicates indicate different control flow taken between two versions under the same input and configuration. Such control flow differences are useful in explaining which part of the program might be behaving unexpectedly.

3.4 Configuration Option Recommendation

In this step, ConfSuggester attributes the control flow differences to one or more root-cause configuration options. The key idea is to identify configuration options that may affect the behaviorally-deviated predicates, and then rank these options by the deviation value (computed by the deviation function in Figure 7) and the number of executed statements they control (computed by the getExecutedStmtNum auxiliary function in Figure 7).

To identify the configuration options that can affect a predicate, a straightforward way is to use program slicing [64] to compute a forward slice from the initialization statement of a configuration option, and then check whether the predicate is in the slice. Unfortunately, traditional full slicing [64] would produce unusably large slices due to its conservatism.

To address this limitation, ConfSuggester uses thin slicing [53] to identify configuration options that *directly* affect a predicate. Different from traditional full slicing, thin slicing *only* follows the data flow dependencies from the slicing criterion (i.e., the initialization statement of a configuration option) and ignores control flow dependencies as well as uses of base pointers. Using thin slicing, ConfSuggester separates pointer computations from the flow of configuration option values and naturally connects a configuration option with its affected statements by the data flow dependencies. Section 5.3.4 empirically demonstrates that using traditional full slicing will decrease the accuracy of ConfSuggester.

Auxiliary functions:
getPredicates(T): return all executed predicates in the execution trace T.
getAffectingOptions(p, V): use thin slicing [53] to compute all configuration options that may affect predicate p in the software version V.
getExecutedStmtNum(p, V, T): return the number of executed statements (controlled by predicate p) in trace T from software version V.

```
deviation(p_1, T_1, p_2, T_2):
 1: result ← |φ(p_1, T_1) − φ(p_2, T_2)|
    {δ is a pre-defined threshold with default value: 0.1}
 2: if result < δ then
 3:     result = 0
 4: end if
 5: return result
```

Input: two software versions: V_{old} and V_{new}.
two execution traces: T_{old} and T_{new}, on the respective versions.
a map of matched statements between V_{old} and V_{new} : *stmtMap*.
Output: a ranked list of likely root-cause configuration options

```
recommendOptions(V_old, V_new, T_old, T_new, stmtMap)
 1: optionMap ← new Map⟨Option, Float⟩
    {Each entry of optionMap is initialized to 0.}
 2: for each p_old in getPredicates(V_old) do
 3:     d ← deviation(p_old, T_old, stmtMap[p_old], T_new)
 4:     options_old ← getAffectingOptions(p_old, V_old)
 5:     w ← d × getExecutedStmtNum(p_old, V_old, T_old)
 6:     for each Option option in options_old do
 7:         optionMap[option] ← optionMap[option] + w
 8:     end for
 9: end for
10: for each p_new in getPredicates(V_new) do
11:     d ← deviation(stmtMap^-1[p_new], T_old, p_new, T_new)
12:     options_new ← getAffectingOptions(p_new, V_new)
13:     w ← d × getExecutedStmtNum(p_new, V_new, T_new)
14:     for each Option option in options_new do
15:         optionMap[option] ← optionMap[option] + w
16:     end for
17: end for
18: return optionMap.sortedKeys()
```

Figure 7: Algorithm for recommending configuration options. Function deviation is a helper function to compute the deviation value between two predicates p_1 and p_2, and function ϕ used in deviation is defined in Section 3.3.2.

To reason about the root-cause configuration options, ConfSuggester associates each configuration option with a weight, which represents the strength of the causal relationship between the configuration option and the execution differences. A larger weight value indicates that a configuration option potentially contributes more to the control flow differences as its value propagates in the program, and thus the configuration option is more likely to be the root cause.

Figure 7 presents the configuration option recommendation algorithm. For each behaviorally-deviated predicate in an execution trace, ConfSuggester first attributes the deviated behavior to its affecting configuration options (lines 4 and 12). Then, ConfSuggester computes the number of executed statements controlled by that predicate (lines 5 and 13). To do so, the getExecutedStmtNum auxiliary function first statically examines the source code to compute the immediate post-dominator statement [61] of a predicate, and then traverses the execution trace to count the number of statements that are executed between the predicate and its post-dominator statement. ConfSuggester multiples a predicate's deviation value by the number of executed statements, and then updates the weight of each affecting configuration option (lines 5–8 and 13–16). Finally, ConfSuggester ranks all affecting configuration options in decreasing order by weight, outputting a ranked list of suspicious options that might be responsible for the behavioral differences (line 18).

If two configuration options have the same weights, ConfSuggester prefers the configuration option affecting more statements in its thin slice. This heuristic is based on the intuition that configuration options affecting more statements seem more likely to be relevant to the behavioral differences.

3.5 Discussion

We next discuss some design issues in ConfSuggester.

Fixing configuration errors vs. Localizing regression bugs. The problem addressed in this paper is significantly different than the traditional regression bug localization problem [71,75]. A regression bug occurs when developers have made a mistake, which causes the software to violate its specification after a session of code changes. By contrast, in our context, the software behavior on the new version is still as *designed* by the developers but *undesired* by the users.

Why not use a dynamic analysis to recommend configuration options? ConfSuggester uses thin slicing to statically identify responsible configuration options for a behaviorally-deviated predicate. An alternative is to use a pure dynamic analysis to assess how a configuration option may affect the control flow. Techniques such as Delta Debugging [71], value replacement [76], and dual slicing [56] use a similar idea: they repeatedly replace a variable value with other alternatives, and then re-execute the program to check whether the outcome is desired. There are two major challenges that prevent these dynamic analyses from being used. First, it can be difficult to find a valid replacement value for a non-Boolean configuration option, such as a string or regular expression. Second, automatically checking program outcomes requires a testing oracle, which is often not available in practice, and end-users should not be expected to provide it. To address these challenges, ConfSuggester approximates the program behavioral differences by the control flow differences of two executions, and then statically reasons about the responsible configuration options.

ConfSuggester's current limitations. There are three major limitations in the our ConfSuggester technique. First, ConfSuggester assumes the different behaviors of two program versions are not caused by non-determinism. For non-deterministic behaviors, ConfSuggester could potentially leverage a deterministic replay system [23,26] to faithfully reproduce the behaviors. Second, ConfSuggester only matches one predicate in the old program version to one predicate in the new program version. If a predicate evolves into multiple predicates in the new version, ConfSuggester may output less useful results. Third, ConfSuggester focuses on identifying root-cause configuration options that can change the functional behaviors of the target program. Configuration options that affect the underlying OS or runtime system, such as the `-Xmx` option used to specify JVM's heap size when launching a Java program, are not supported by ConfSuggester.

4. IMPLEMENTATION

ConfSuggester uses the WALA framework [62] to perform offline bytecode instrumentation. The instrumentation code records the execution of every statement and the evaluation result of each predicate. ConfSuggester also uses WALA to analyze Java bytecode statically to identify the affecting configuration options for each predicate that behaves differently across versions.

Program	Old Version	New Version	LOC (new version)	ΔLOC	#Options
Randoop	1.2.1	1.3.2	18571	1893	57
Weka	3.6.1	3.6.2	275035	1458	14
Synoptic	0.05	0.1	19153	1658	37
JChord	2.0	2.1	26617	3085	79
JMeter	2.8	2.9	91979	3264	55
Javalanche	0.36	0.40	25144	9261	35

Figure 8: All subject programs used in the evaluation. Column "ΔLOC" shows the number of changed lines of code between the old and new versions. Column "#Options" shows the number of configuration options supported in the new program version.

Like other existing configuration error diagnosis tools [47, 74], ConfSuggester does not instrument libraries such as the JDK, since a configuration option set in the client software usually does not affect the behaviors of its dependent libraries.

5. EVALUATION

We evaluated 4 aspects of ConfSuggester's effectiveness, answering the following research questions:

1. How accurate is ConfSuggester in identifying the root-cause configuration options? That is, what is the rank of the actual root-cause configuration option in ConfSuggester's output (Section 5.3.1)?
2. How long does it take for ConfSuggester to diagnose a configuration error (Section 5.3.2)?
3. How does ConfSuggester's effectiveness compare to existing approaches (Section 5.3.3)?
4. How does ConfSuggester's effectiveness compare to two variants? The first variant uses full slicing in identifying suspicious configuration options, and the second variant only uses predicate behavior changes to recommend configuration options (Section 5.3.4).

5.1 Subject Programs

We evaluated ConfSuggester on 6 Java programs listed in Figure 1. The first 5 subject programs are the 5 Java programs studied in Section 2, and the remaining subject program is Javalanche [24], which is a mutation testing framework.

We included Javalanche because one of its real users provided us a configuration error he encountered when using Javalanche.

5.1.1 Configuration Errors

For the 5 Java programs studied in Section 2, we manually examined all deleted and modified configuration options listed in Figure 2. (The added configuration options are unlikely to cause a misconfiguration.) For each change, based on our own understanding, we wrote a test driver to cover it, and then checked whether the test driver could reveal different behaviors on two versions. For those 5 programs, we collected 7 errors as listed in Figure 9 (the first 7 errors). For the Javalanche program, we reproduced the reported configuration error. In Figure 9, errors #3 and #4 can be reproduced together in a single execution, and each of the other errors is reproduced in one execution.

Our methodology of collecting configuration errors is different from what was used in collecting software regression bugs in the literature [71, 75]. Software regression bugs often can be found in well-maintained bug databases. By contrast, finding recorded configuration errors is much harder, mainly because most configuration errors have not been documented rigorously [69]. Usually, after a session of code changes, when regression tests pass, developers may treat the software behaviors as having been validated. Further, because the software misconfigurations are user-driven, the "fixes" may be recorded simply as pointers to manuals or other documents.

5.2 Evaluation Procedure

For each subject program, we used ConfSuggester to instrument both versions. For each configuration error, we used the same input and configuration to reproduce the different behaviors on two instrumented versions.

The average size of the execution traces is 40MB, and the largest one (Randoop's trace) is 140MB.

When using ConfSuggester to diagnose a configuration error, we manually specify the initialization statement of each configuration option as the thin slicing criterion. This manual, one-time-cost step took 20 minutes on average per subject program. After that, ConfSuggester works in a fully-automatic way: it analyzes two program versions and two execution traces, and outputs a ranked list of configuration options. Future work should automate this manual step.

Our experiments were run on a 2.67GHz Intel Core PC with 4GB physical memory (2GB was allocated for the JVM), running Windows 7.

5.3 Results

5.3.1 Accuracy

As shown in Figure 9, ConfSuggester is highly effective in identifying the root-cause configuration options that should be changed in the new program version. The average rank of the root cause in ConfSuggester's output is 1.8. For 6 errors, the root-cause configuration option ranks first in ConfSuggester's output; for 1 error, the root-cause configuration option ranks third in ConfSuggester's output; and the root-cause option ranks sixth for the remaining error. ConfSuggester is successful because of its ability to identify the behaviorally-deviated predicates with substantial impacts through execution trace comparison. The top-ranked deviated predicates often provide useful clues about what parts of a program have performed differently.

Summary. ConfSuggester recommends correct configuration options with high accuracy for evolving configurable software systems with non-trivial code changes.

5.3.2 Performance of ConfSuggester

We measured ConfSuggester's performance in two ways: the performance overhead introduced by instrumentation when demonstrating the configuration error, and the time cost of recommending configuration options. Figure 10 shows the results.

The performance overhead to demonstrate the error varies among programs. The current implementation imposes an average 8× and 12.8× slowdown in a ConfSuggester-instrumented old and new program version, respectively. This is due to ConfSuggester's inefficient instrumentation code that monitors the execution of every instruction. The overhead could be reduced by instrumenting at basic block granularity instead. Even so, except for two errors (errors #5 and #6) in JChord, all other errors can be reproduced in less than 30 seconds. Errors #5 and #6 require about 20 minutes to reproduce.

ConfSuggester spends an average of 3.1 minutes to recommend configuration options for one error (including the time to compute thin slices and the time to suggest suspicious options). Computing thin slices for all configuration options is non-trivial. However, this step is one-time cost per program and the results can be precomputed. The time used for suggesting configuration options is roughly

Error ID. Program	Error Description	Root-Cause Configuration Option	#Options	Rank of the Root-Cause Configuration Option		
				ConfSuggester	ConfDiagnoser [74]	ConfAnalyzer [47]
1. Randoop	Poor performance in test generation	`usethreads`	57	1	**N**	**X**
2. Weka	A different error message when Weka crashes	`m_numFolds`	14	1	9	1
3. Synoptic	Initial model not saved	`dumpInitialGraphDotFile`	37	1	**N**	**X**
4. Synoptic	Generated model not saved as JPEG file	`dumpInitialGraphPngFile`	37	6	**N**	**X**
5. JChord	Bytecode parsed incorrectly	`chord.ssa`	79	1	3	**X**
6. JChord	Method names not printed in the console	`chord.print.methods`	79	1	**N**	**X**
7. JMeter	Results saved to a file with a different format	`output_format`	55	1	1	**X**
8. Javalanche	No mutants generated	`project.tests`	35	3	4	**X**
Average			49.1	1.8	15.3	47.5

Figure 9: All configuration errors used in the evaluation and the experimental results. Only the 2nd error is a crashing error, and all the other errors are non-crashing errors. Column "Root-Cause Configuration Option" shows the actual root-cause configuration option. Column "#Options" shows the number of configuration options supported in the new program version, taken from Figure 8. Column "Rank of the Root-Cause Configuration Option" shows the absolute rank of the actual root-cause configuration option in each technique's output (lower is better). "X" means the technique is not applicable (i.e., requiring a crashing point), and "N" means the technique does not identify the actual root cause. When computing the average rank, each "X" or "N" is treated as half of the number of configuration options, because a user would need to examine on average half of the available options to find the root cause. Column "ConfSuggester" shows the results of using our technique. Columns "ConfDiagnoser" and "ConfAnalyzer" show the results of using two existing techniques as described in Section 5.3.3.

Error ID. Program	Run-time Slowdown (×)		ConfSuggester time (s)	
	Old Version	New Version	Slicing	Suggestion
1. Randoop	20.1	4.1	90	295
2. Weka	1.6	1.6	80	49
3. Synoptic	1.7	4.7	48	42
4. Synoptic	1.7	4.7	48	42
5. JChord	18.7	44.3	20	38
6. JChord	17.6	41.1	23	29
7. JMeter	1.3	1.4	51	63
8. Javalanche	1.4	1.5	430	265
Average	8.0	12.8	99	91

Figure 10: ConfSuggester's performance. The "Run-time Slowdown" column shows the cost of reproducing the error in an ConfSuggester-instrumented version of the subject program. The "ConfSuggester time (s)" column shows the time taken by ConfSuggester to diagnose configuration errors in seconds. Column "Slicing" is the cost of computing thin slices on both old and new program versions.

proportional to the size of the execution trace rather than the size of the subject program.

Summary. ConfSuggester recommends configuration options for diagnosing configuration errors with reasonable time cost.

5.3.3 Comparison with Two Existing Approaches

This section compares ConfSuggester with two existing approaches, ConfDiagnoser [74] and ConfAnalyzer [47]. ConfDiagnoser and ConfAnalyzer are among the most precise configuration error diagnosis techniques in the literature.

ConfDiagnoser, proposed in our previous work [74], is an automated software configuration error diagnosis technique. ConfDiagnoser is *not* cognizant of software evolution, and it diagnoses configuration errors from a single program version. ConfDiagnoser assumes the existence of a set of correct execution traces, which are used to compare against the undesired execution trace to identify the abnormal program parts. When comparing the undesired execution trace with a correct execution trace, ConfDiagnoser only uses a predicate's deviation value to reason about the most suspicious options, while ignoring the statements controlled by a predicate's evaluation result.

To compare ConfSuggester with ConfDiagnoser, we reused the pre-built execution trace databases for the 4 shared subject programs (Randoop, Synoptic, JChord, and Weka) from [74]. Each existing trace database contains 6–16 correct execution traces. For the remaining two subject programs (JMeter and Javalanche), we manually built an execution trace database for each of them by running correct examples from their user manuals. The databases contain 6 and 8 execution traces for JMeter and Javalanche, respectively.

ConfAnalyzer, proposed by Rabkin and Katz [47], is a lightweight static configuration error diagnosis technique. ConfAnalyzer tracks the flow of labeled objects through program control flow and data flow, and treats a configuration option as a root cause if its value may flow to a crashing point. Since ConfAnalyzer cannot diagnose non-crashing errors, we can only apply it to diagnose the crashing error in Weka (error #2 in Figure 9).

Results. Columns "ConfDiagnoser" and "ConfAnalyzer" in Figure 9 show the experimental results.

ConfSuggester produces significantly more accurate results than ConfDiagnoser, primarily for two reasons. First, ConfDiagnoser focuses on diagnosing *erroneous* program behaviors and identifies their responsible configuration options. However, for the problem addressed in this paper, the new software version that exhibits *undesired* behavior (after applying the same configuration used in the old version) is working exactly as *designed*. In other words, the execution trace obtained by running the new program version is still *correct*. Therefore, just comparing execution traces obtained from the new program version is not effective in identifying the "abnormal" behavior. By contrast, ConfSuggester compares execution traces from two different versions and directly reasons about the execution differences. Second, ConfDiagnoser only focuses on the predicate behavior changes, while ignoring the statements potentially impacted by the affected predicate. This makes ConfDiagnoser fail to distinguish predicates whose behavioral changes can have different impacts. Section 5.3.4 further evaluates this design choice, showing that considering the number of controlled statements can substantially increase the diagnosis accuracy.

ConfAnalyzer outputs the correct result for the crashing error in Weka, but cannot identify root causes for other non-crashing errors. The crashing error in Weka occurs soon after the program is launched. ConfAnalyzer correctly identifies its root cause because a small number of configuration options are initialized and only one of them flows to the crashing point.

ConfSuggester is not directly comparable to other related configuration error diagnosis approaches [4, 6, 55, 63, 66, 68]. Existing approaches target a rather different problem than ConfSuggester,

Error ID.	Rank of the Root-Cause Configuration Option		
Program	ConfSuggester	Full Slicing	Predicate Behavior
1. Randoop	1	32	7
2. Weka	1	7	1
3. Synoptic	1	16	3
4. Synoptic	6	17	8
5. JChord	1	19	5
6. JChord	1	30	5
7. JMeter	1	**N**	1
8. Javalanche	3	**N**	13
Average	1.8	20.7	5.4

Figure 11: Experimental results of evaluating two design choices of ConfSuggester. Column "ConfSuggester" shows ConfSuggester's results, taken from Figure 9. Column "Full Slicing" shows the results of replacing thin slicing with full slicing in ConfSuggester. "N" means the technique does not identify the actual root cause. Column "Predicate Behavior" shows the results of ConfSuggester, if it only considers predicate behavior change. When computing the average rank, each "N" is treated as half of the number of configuration options.

or require different inputs than ConfSuggester. For example, X-Ray [4] diagnoses configuration errors on a single program version. PeerPressure [63] and RangerFixer [68] only support configuration options defined by certain specific feature models. General software fault localization techniques [28, 37] are not well-suited for configuration error diagnosis, since such techniques often focus on identifying the buggy code or invalid input values. This has been empirically validated in our previous work [74].

Summary. Configuration error diagnosis techniques designed for a *single* program version achieve less accurate results in diagnosing configuration errors introduced in software evolution. ConfSuggester reasons about the behavioral differences between two program versions, and produces more accurate results.

5.3.4 Evaluating Two Design Choices

This section evaluates two design choices in ConfSuggester.

Slicing algorithms. ConfSuggester uses thin slicing to identify configuration options whose values may affect a predicate. We next evaluate a variant that replaces thin slicing with the traditional full slicing [22]. This variant changes the getAffectingOptions auxiliary function in Figure 7, by using full slicing to compute all configuration options that may affect a predicate. Figure 11 (Column "Full Slicing") shows the results.

ConfSuggester achieves substantially less accurate results when using full slicing. The primary reason is that full slicing identifies many irrelevant configuration options that *indirectly* affect a predicate of interest. Such configuration options are not pertinent to the task of error diagnosis. Linking them to the exhibited different behavior would degrade ConfSuggester's accuracy. Further, computing full slices is much more expensive than computing thin slices. WALA's full slicing algorithm failed to scale to two subject programs (JMeter and Javalanche).

Predicate behavioral change metrics. ConfSuggester considers both the predicate behavior change and the number of affected statements in diagnosing configuration errors. We next evaluate a variant that only uses the predicate behavior change to diagnose errors. This variant changes the getExecutedStmtNum auxiliary function in Figure 7, by making it always return 1. Figure 11 (Column "Predicate Behavior") shows the results.

ConfSuggester's accuracy degrades substantially when ranking predicates based on its behavioral changes without considering the number of affected statements. The primary reason is that behaviorally-deviated predicates occur all over the execution traces, but each predicate may have different impacts to the overall program behavior change. ConfSuggester uses the number of statements determined by the predicate evaluation result to approximate such potential impacts.

Summary. Full slicing includes too many irrelevant program statements due to its conservatism and only using a predicate's behavior change is not enough to identify the root-cause configuration options. ConfSuggester, using thin slicing and considering both the predicate behavior change and the impacted statements, is a better choice in diagnosing configuration errors.

5.4 Discussion

Threats to validity. There are several threats to validity of our evaluation. First, the 6 Java programs might not be representative, though some of them have been used in previous research. Likewise, the 8 configuration errors might not be representative, even though we evaluated every error we found. We only evaluated ConfSuggester on errors caused by one configuration option. It is unclear whether ConfSuggester would produce useful results if fixing a particular configuration error requires changing values of two dependent configuration options. Second, our evaluation focused on configuration errors rather than software regression bugs, as all regression tests between two versions pass. We have not evaluated whether ConfSuggester would help users work around buggy program versions. Third, ConfSuggester's effectiveness depends on the effectiveness of the predicate-matching algorithm. In our experiments, on average 12% of lines are changed between the program versions. ConfSuggester may yield less useful results for programs with significant code changes. However, different algorithms can be plugged into ConfSuggester. Fourth, our evaluation only compared ConfSuggester with two other approaches. Comparing with other analyses or tools might yield different observations.

Experimental conclusions. We have three chief findings. (1) ConfSuggester is highly effective in diagnosing configuration errors introduced by software evolution; (2) ConfSuggester produces more accurate results than approaches designed to diagnose errors on a single program version; and (3) ConfSuggester outperforms two variants that use full slicing and only a predicate's behavior change in error diagnosis, respectively.

6. RELATED WORK

The most closely related work falls into three categories: (1) techniques for supporting software evolution; (2) software configuration error diagnosis techniques; and (3) configuration-aware software analysis techniques.

6.1 Supporting Software Evolution

As software evolves, its behavior must be validated. Regression test selection [19] indicates which tests need to be executed for a changed program. Program differencing techniques [11, 13, 18, 26, 29, 33, 43, 60, 67] identify changes between two program versions and present the change list to developers for inspection. Change impact analysis techniques [35], which are often built on top of program differencing techniques, identify not only the changes, but also code fragments that are affected by the changes. Different than ConfSuggester's predicate-matching algorithm (Section 3.3.1), existing program differencing techniques primarily focus on matching program elements at the method level [11, 13, 29, 33, 39, 43, 67, 72], or matching program statements on the source code based on textual similarity [21]. By contrast, ConfSuggester's matching algorithm, inspired by the JDiff algorithm [3], is specifically designed to match the evolved predicate in the new program version. (See Section 3.3.1 for a detailed comparison with JDiff.) The algorithm directly works on the bytecode of two program versions without any additional information from users, such as a software revision history [39].

Nagarajan et al. [42] developed a technique to match control flows of two program versions running with the same input. Different from ConfSuggester, their work assumes semantically-equivalent program versions (e.g., optimized and unoptimized), while ConfSuggester compares two versions that include functional changes.

Many techniques have been developed to identify failure-inducing code changes for evolving software [7, 20, 36, 44]. For example, Delta Debugging aims to find a minimal subset of changes that still makes the test fail [71]. Test minimization techniques [20, 73] simplify the failed test to ease comprehension for developers. ConfSuggester differs from these techniques in three aspects. First, existing techniques focus on helping software developers localize a bug, while ConfSuggester targets software configuration errors fixable by software end-users. As we have discussed in Section 3.5, configuration errors are fundamentally different than regression bugs. They are mostly user-driven and do not indicate problems in the source code. Second, most of the existing techniques identify *what* (e.g., a snippet of code) causes the regression bug, but do not answer the question of *how* (e.g., which configuration option should a user change?) to fix the error. By contrast, ConfSuggester explicitly guides users to suspicious configuration options. Third, most of the regression failure localization techniques [71] require a testing oracle for automated correctness checking. However, such oracles are often absent in practice. By contrast, ConfSuggester eliminates this requirement by approximating the software behavioral difference as the control flow differences.

6.2 Software Configuration Error Diagnosis

Software configuration errors are time-consuming and frustrating to diagnose. To reduce the time and human effort needed to troubleshoot software misconfigurations, prior research has applied different techniques to the problem of configuration error diagnosis [5, 6, 31, 47, 63, 66, 68]. For example, Chronus [66] relies on a user-provided testing oracle to check the system behavior, and uses virtual machine checkpoint and binary search to find the point in time where the program behavior switched from correct to incorrect. AutoBash [55] fixes a misconfiguration by using OS-level speculative execution to try possible configurations, examine their effects, and roll them back when necessary. PeerPressure [63] statistically compares configuration states in the Windows Registry on different machines. When a registry entry value on a machine exhibiting erroneous behavior differs from the value usually chosen by other machines, PeerPressure flags the value as a potential error. More recently, ConfAid [6] and X-Ray [4] use dynamic taint analysis to diagnose configuration errors by monitoring causality within the program binary as it executes. ConfAnalyzer [47] uses dynamic information flow analysis to precompute possible configuration error diagnoses for every possible crashing point in a program.

ConfSuggester is significantly different from the existing approaches. First, ConfSuggester is cognizant of software evolution while most previous approaches are not [5, 6, 47, 66]. Second, ConfSuggester supports diagnosing both crashing and non-crashing errors while most techniques can only diagnose configuration errors that lead to a crash or assertion failure [5, 6, 47, 66]. Third, unlike several approaches [6, 66], ConfSuggester does not assume the existence of a testing oracle. Fourth, ConfSuggester uses platform-independent offline instrumentation and requires no alternation to the underlying operating system or runtime environment. This differs from existing OS-level diagnosis techniques [55, 66]. Fifth, approaches like PeerPressure [63] and RangeFixer [68] benefit from the known schema of the Windows Registry and feature models, but cannot diagnose configuration errors that lie outside these specific domains. Our technique of analyzing the execution traces is more general.

6.3 Configuration-Aware Software Analysis

Software configuration management is a central component of software product lines. Many configuration-aware software analysis techniques have been developed to analyze configurable software systems [9, 30, 34, 38], improve software configuration management [8, 10, 17, 46, 59], and understand and test the behavior of a configurable software system [2, 32, 45, 50, 51, 54].

Compared to ConfSuggester, these techniques have rather different goals. They primarily focus on reducing the burden of configuration management and preventing certain errors from happening, or creating test suites to find new errors in a configurable software system earlier. They cannot diagnose an exhibited configuration error during software evolution. By contrast, ConfSuggester links the behavioral differences to a small number of configuration options and explicitly guides software end-users to the root causes.

7. CONCLUSION AND FUTURE WORK

This paper describes ConfSuggester, a technique to help software users to troubleshoot configuration errors. ConfSuggester focuses on errors caused by software evolution, and recommends configuration options whose values should be changed to produce the desired behavior on the new software version. In our experiments, ConfSuggester accurately identified the root causes of 8 configuration errors in 6 real-world software systems. The source code of ConfSuggester is publicly available at: `http://config-errors.googlecode.com`.

As future work, we plan a user study to evaluate ConfSuggester's usefulness to end-users. A challenge will be finding study participants who are familiar with only the old versions of given subject programs. We also plan to develop techniques to automatically distinguish software bugs from configuration errors, when a software system exhibits undesired behavior. Such techniques can help formulate guidance regarding when the user should give up on ConfSuggester and assume the error is not related to configuration.

8. ACKNOWLEDGMENTS

We thank Lingming Zhang for providing the configuration error in Javalanche and helpful discussion about using Javalanche. This work was supported in part by NSF grants CCF-1016701 and CCF-0963757.

9. REFERENCES

[1] Apache Server. `http://httpd.apache.org/`.
[2] S. Apel, C. Kastner, and C. Lengauer. FEATUREHOUSE: Language-independent, automated software composition. In *ICSE*, 2009.
[3] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *ASE*, 2004.
[4] M. Attariyan, M. Chow, and J. Flinn. X-ray: automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.
[5] M. Attariyan and J. Flinn. Using causality to diagnose configuration bugs. In *USENIX ATC*, 2008.

[6] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.
[7] A. Banerjee, A. Roychoudhury, J. A. Harlie, and Z. Liang. Golden implementation driven software debugging. In *FSE*, 2010.
[8] J. Barreiros and A. Moreira. A model-based representation of configuration knowledge. In *FOSD*, 2009.
[9] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPLLIFT: statically analyzing software product lines in minutes instead of years. In *PLDI*, 2013.
[10] D. Cooray, S. Malek, R. Roshandel, and D. Kilgore. Resisting reliability degradation through proactive reconfiguration. In *ASE*, 2010.
[11] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE*, 2008.
[12] C. R. B. de Souza and D. F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *ICSE*, 2008.
[13] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, 2006.
[14] R. Johnson. More details on today's outage. `https://www.facebook.com/notes/facebook-engineering/more-details-on-todaysoutage/431441338919`.
[15] Volatile and Decentralized. `http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html`.
[16] FireFox. `http://www.mozilla.org/en-US/firefox/new/`.
[17] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Using feature locality: can we leverage history to avoid failures during reconfiguration? In *ASAS*, 2011.
[18] O. Giroux and M. P. Robillard. Detecting increases in feature coupling using regression tests. In *FSE*, 2006.
[19] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *OOPSLA*, 2001.
[20] K. J. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware trace analysis. In *PLDI*, 2009.
[21] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI*, 1990.
[22] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI*, 1988.
[23] J. Huang, C. Zhang, and J. Dolby. CLAP: recording local executions to reproduce concurrency failures. In *PLDI*, 2013.
[24] Javalanche. `https://github.com/david-schuler/javalanche/`.
[25] JChord. `http://pag.gatech.edu/chord/`.
[26] W. Jin and A. Orso. BugRedux: reproducing field failures for in-house debugging. In *ICSE*, 2012.
[27] JMeter. `http://jmeter.apache.org/`.
[28] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, 2002.
[29] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.
[30] K. C. Kang, M. Kim, J. Lee, and B. Kim. Feature-oriented re-engineering of legacy systems into product line assets: a case study. In *SPLC*, 2005.
[31] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *DSN*, 2008.
[32] C. H. P. Kim, D. Marinov, S. Khurshid, and D. Batory. SPLat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *FSE*, 2013.
[33] M. Kim, D. Notkin, D. Grossman, and G. Wilson Jr. Identifying and summarizing systematic code changes via rule inference. *IEEE Trans. Softw. Eng.*, 39(1):45–62, Jan. 2013.
[34] I. H. Krüger, R. Mathew, and M. Meisinger. From scenarios to aspects: exploring product lines. In *SCESM*, 2005.
[35] B. Li, X. Sun, H. Leung, and S. Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 2012.
[36] C. Liu, J. Yang, L. Tan, and M. Hafiz. R2Fix: Automatically generating bug fixes from bug reports. In *ICST*, 2013.
[37] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *ESEC/FSE*, 2003.
[38] T. Mende, F. Beckwermert, R. Koschke, and G. Meier. Supporting the grow-and-prune model in software product lines evolution using clone detection. In *CSMR*, 2008.
[39] S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In *ICSE*, 2012.
[40] Configuration error brings down the Azure cloud platform. `http://www.evolven.com/blog/configuration-error-brings-down-the-azure-cloud\-platform.html`.
[41] MySQL. `http://www.mysql.com/`.
[42] V. Nagarajan, R. Gupta, X. Zhang, M. Madou, and B. De Sutter. Matching control flow of program versions. In *2007*, 2007.
[43] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *ICSE*, 2010.
[44] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: an approach for debugging evolving programs. In *FSE*, 2009.
[45] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *ISSTA*, 2008.
[46] R. Rabiser, P. Grünbacher, and M. Lehofer. A qualitative study on user guidance capabilities in product configuration tools. In *ASE*, 2012.
[47] A. Rabkin and R. Katz. Precomputing possible configuration error diagnoses. In *ASE*, 2011.
[48] Randoop. `http://code.google.com/p/randoop/`.
[49] Misconfiguration brings down entire .se domain in Sweden. `http://www.circleid.com/posts/misconfiguration_brings_down_entire_se_domain_in_sweden`.
[50] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *ICSE*, 2013.
[51] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *ICSE*, 2012.
[52] Y. J. Song, F. Junqueira, and B. Reed. Bft for the skeptics. In *Proc. ACM SOSP'09 Work in Progress Session*.
[53] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI*, 2007.
[54] M. Staats, M. W. Whalen, and M. P. Heimdahl. Programs,

tests, and oracles: the foundations of testing revisited. In *ICSE*, 2011.

[55] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: improving configuration management with operating system causality analysis. In *SOSP*, 2007.

[56] W. N. Sumner and X. Zhang. Comparative causality: explaining the differences between executions. In *ICSE*, 2013.

[57] Synoptic. `http://code.google.com/p/synoptic/`.

[58] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: an exploratory study in industry. In *FSE*, 2012.

[59] M. H. Ter Beek, H. Muccini, and P. Pelliccione. Guaranteeing correct evolution of software product lines: setting up the problem. In *SERENE*, 2011.

[60] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Softw. Engg.*, 15(1), Feb. 2010.

[61] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[62] WALA. `http://sourceforge.net/projects/wala/`.

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

[64] M. Weiser. Program slicing. In *ICSE*, 1981.

[65] Weka. `www.cs.waikato.ac.nz/ml/weka/`.

[66] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI*, 2004.

[67] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *ASE*, 2005.

[68] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In *ICSE*, 2012.

[69] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, 2011.

[70] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, Sept. 2004.

[71] A. Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Softw. Eng. Notes*, 24(6):253–267, 1999.

[72] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *ICSM*, 2011.

[73] S. Zhang. Practical semantic test simplification. In *ICSE (NIER Track)*, 2013.

[74] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *ICSE*, 2013.

[75] S. Zhang, Y. Lin, Z. Gu, and J. Zhao. Effective identification of failure-inducing changes: a hybrid approach. In *PASTE*, 2008.

[76] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *ASE*, 2011.

[77] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE*, 2004.