# Context-based Online Configuration-Error Detection

*Ding Yuan*[1,2]*, *Yinglian Xie*[3], *Rina Panigrahy*[3], *Junfeng Yang*[4]*, *Chad Verbowski*[5], *Arunvijay Kumar*[5]
[1]*University of Illinois at Urbana-Champaign,* [2]*University of California, San Diego*
[3]*Microsoft Research Silicon Valley,* [4]*Columbia University,* [5]*Microsoft Corporation*

## Abstract

Software failures due to configuration errors are commonplace as computer systems continue to grow larger and more complex. Troubleshooting these configuration errors is a major administration cost, especially in server clusters where problems often go undetected without user interference.

This paper presents CODE–a tool that automatically detects software configuration errors. Our approach is based on identifying invariant configuration access rules that predict what access events follow what contexts. It requires no source code, application-specific semantics, or heavyweight program analysis. Using these rules, CODE can sift through a voluminous number of events and detect deviant program executions. This is in contrast to previous approaches that focus on only diagnosis. In our experiments, CODE successfully detected a real configuration error in one of our deployment machines, in addition to 20 user-reported errors that we reproduced in our test environment. When analyzing month-long event logs from both user desktops and production servers, CODE yielded a low false positive rate. The efficiency of CODE makes it feasible to be deployed as a practical management tool with low overhead.

## 1 Introduction

Software configuration errors impose a major cost on system administration. Configuration errors may result in security vulnerabilities, application crashes, severe disruptions in software functionality, unexpected changes in the UI, and incorrect program executions [7]. While several approaches have attempted to automate configuration error diagnosis [2, 20, 26, 29], they rely solely on manual efforts to detect the error symptoms [20, 26, 29]. As usual, manual approaches incur high overhead (e.g., requiring users to write error-detection scripts for each application) and are unreliable (e.g., security policy errors may show no user-visible symptoms). These drawbacks often lead to long delays between the occurrence and the detection of errors, causing unrecoverable damage to system states.

In this paper, we aim to automatically detect configuration errors that are triggered by changes in configuration data. These types of errors are commonplace and can be introduced in many ways, such as operator mistakes, software updates or even software bugs that corrupt configuration data. For example, a software update may turn off the "AutoComplete" option for a Web browser, which, as a result, can no longer remember usernames or passwords. An accidental menu click by a user may corrupt a configuration entry and cause an application toolbar to disappear. A seemingly benign user operation that disables the ActiveX control can unexpectedly disable the remote desktop application.

We consider configuration data because it captures important OS and application settings. Further, the data is typically accessed through well defined interfaces such as Windows Registries. We can thus treat the applications and the OS as black boxes, *transparently* intercepting and checking configuration accessing events (called *events* hereafter). This approach is lightweight: it does not require modifying the OS [13] or using virtual machines [29].

We focus on Windows, where applications use the Registry to store and access configuration data. In particular, we log all Registry events and analyze them online to automatically detect errors. While Windows has the largest OS market share [1] and is also the focus of many previous efforts [26], our methodologies can be generalized to other types of OS and configuration data.

Analyzing configuration-access events automatically for error detection faces three practical challenges. First, we need to efficiently process a huge number of events. A typical Windows machine has on average 200 thousand Registry entries [26], with $10^6$ to $10^8$ access events per day [23]. Commonly used learning techniques (e.g., [1, 28]) rarely scale to this level.

Second, we must automatically handle a large set of diverse applications. Different applications may have drastically different configuration access patterns. These patterns may evolve with user behavior changes or software updates.

Finally, our analysis must effectively detect errors without generating a large number of false positives. Configuration data is highly dynamic: there are, on average, $10^4$ writes to Registry per day per machine, and $10^2$ of them are writes to frequently accessed Registries

*This work was done when the authors were at Microsoft Research Silicon Valley.

[1]Specifically, Windows has 91% of client operating system market [22, 31] and 74% of server market [10].

that have never changed before. Application runtime behaviors such as user inputs, caching, and performance optimizations may all add noise and unpredictability to configuration states, making it difficult to distinguish between real errors and normal updates.

In this paper we present CODE, an automatic online configuration-error detection tool for system administrators. CODE is based on the observation that the seemingly unrelated events are actually dependent. The events externalize the control flow of a program and typically occur in predictable orders. Therefore, a sequence of events provides the *context* of a program's runtime behavior and often implies what follows. Further, the more frequently a group of events appear together, the more correlated they should be.

Thus, rather than analyzing each event in isolation, CODE extracts *repetitive*, *predictable* event sequences, and constructs invariant configuration access rules in the form of $context \rightarrow event$ that a program should follow. CODE then enforces these rules and reports out-of-context events as errors. By tracking sequences, CODE also enables richer error diagnosis than looking at each individual event. Once CODE detects an error, it also suggests a possible fix based on the context, the expected event, and the error event.

We implemented CODE as a stand-alone tool that runs continuously on a single desktop for error detection. It can also be extended to support centralized configuration management in data center environments. Our evaluation, using both real user desktops and production servers, shows that the context-based approach has four desirable features:

- *Application independent:* CODE requires no source code, application semantics, or heavyweight program analysis to generate contexts; it can automatically construct rules to represent more than 80% of events for most processes we studied.
- *Effective:* CODE successfully detected all reproduced real-world configuration errors and 96.6% of randomly injected errors in our experiments. CODE also detected a real configuration error on a coauthor's desktop.
- *Configurable false positive rate:* Since CODE reports only out-of-context events instead of new events, it will not report normal configuration changes as alarms. Further, the false positive rate is configurable. In our experiments it reports an average of 0.26 warning per desktop per day and 0.06 per server per day.
- *Low overhead:* CODE keeps only a small number of rules for detection and processes events as they arrive online. The CPU overhead is small (less than 1% over 99% of the time). The memory overhead is less than 0.5% for data-center servers with 16GB memory.

We explicitly designed CODE to detect *configuration* errors; our goal is *not* to catch all errors or malicious attacks. We view our focus on frequent event sequences as a good tradeoff. The high access frequencies indicate that errors in these events are more critical. Moreover, our detection takes place at the time when erroneous configurations are accessed and manifest. Hence, these errors are the ones that actually affected normal program executions, and CODE naturally concentrates on them.

This paper is organized as follows. We first discuss related work in Section 2 and introduce Windows Registry and a motivation example in Section 3. We then present an overview of CODE in Section 4. We next describe its rule learning (Section 5) and error detection (Section 6). We show our evaluation results in Section 7. Finally, we discuss our limitations and future work (Section 8) before we conclude (Section 9).

## 2 Related Work

To our best knowledge, CODE is the first automatic system for online configuration error detection. Below we discuss related work on configuration-error diagnosis and sequence-analysis based intrusion detection.

**Configuration error diagnosis.** Several diagnosis tools have been developed to assist administrators in diagnosing software failures. ConfAid [2] uses information-flow tracking to analyze the dependencies between the error symptoms and the configuration entries to identify root causes. Autobash [20] leverages OS-level speculative execution to causally track activities across different processes. Chronus [29] uses virtual machine checkpoints to maintain a history of the entire system states. KarDo [14] automatically applies the existing fix to a repeated configuration error by searching for a solution in a database. SherLog [33] uses static analysis to infer the execution path based on the runtime log messages to diagnose failures.

Another family of tools compares the configuration data in a problematic system with those in other systems to pinpoint the root cause of a failure [12, 26, 27]. They focus on the snapshots of configuration states, and use statistical tools to compare either historical snapshots or snapshots across machines. While it may seem feasible to extend these state-based approaches for error detection, our experiments showed that such approaches will generate a large number of false positives due to the noise in configuration states (e.g., constant state modifications or legitimate updates). In contrast, CODE reasons about *actions* rather than states for error detection.

The existing systems discussed so far have enhanced off-line diagnosis of configuration errors. However, they all require users or administrators to detect configuration errors. In contrast, CODE focuses on automatic error detection (it can further aid error diagnosis). The importance of having an automatic detection system is also recognized in [19]. Due to the complex dependencies of modern computer systems, detecting faulty configuration states as early as possible helps to isolate the damage and localize the root cause of a failure, especially in server clusters or data centers with thousands of user-unmonitored machines.

**Software resilience to configuration errors** Candea et al. proposed a tool called ConfErr for measuring a system's resilience to configuration errors [4, 11]. ConfErr automatically generates configuration mistakes using human error models rooted in psychology and linguistics. ConfErr and CODE differ in their purposes. ConfErr can help improve software resilience to configuration errors and thus prevent errors from occurring, while CODE can be used to detect and diagnose configuration errors once they occur and is thus complementary.

**Sequence analysis.** A large number of intrusion detection systems (IDS) identify intrusions with abnormal system call sequences (e.g., [6, 9, 24, 32]). They construct models of legal system call sequences by analyzing either the source code or the normal executions in an off-line learning phase. A deviation from the learned models is flagged as an intrusion.

By analyzing event sequences to identify predictable patterns, CODE shares similar benefits to run-time system-call analysis. However, our focus on configuration events instead of system calls leads to significantly different design decisions. Configuration access patterns constantly evolve, so off-line analysis used in IDS systems risks overfitting and producing outdated rules. Further, while IDS systems have to prevent sophisticated attacks [25] using conservative, non-deterministic models, CODE explicitly focuses on the potentially more critical frequent sequences using simple, deterministic rules. More importantly, the heavyweight learning algorithms that IDS systems commonly use make them difficult to scale to the volume of configuration access events, thus these systems are often unable to adapt to dynamic environments online. In contrast, the focus of identifying only invariant rules enables CODE to adopt and adapt much more efficient sequence-analysis methods to operate online.

Prior work (e.g., [8, 15]) has also used event transitions to build program behavior profiles. They mostly focus on depth-2 transitions on code call graphs. In contrast, CODE's event transition rules can consist of all possible lengths of prefixes, thus are more flexible and expressive when representing event sequences as contexts.

## 3 Background and A Motivating Example

In this section, we first introduce Windows Registry, the default configuration store for Windows applications. We then present a motivating configuration-error example and show how CODE can automatically detect and diagnose this error using contexts.

| | |
|---|---|
| Key: | HKEY_LOCAL_MACHINE\Software\Perl |
| Value: | BinDir |
| Data: | C:\Perl\bin\perl.exe |
| Operation: | QueryValue |
| Status: | Success |

Table 1: An example Windows Registry operation.

| | Average | Maximum |
|---|---|---|
| Data modification | 1051 | 5505 |
| Key/Value creation | 883 | 32676 |
| Key/Value deletion | 172 | 4997 |
| Total | 2106 | 43178 |

Table 2: Average and maximum number of Registry update operations/process/day (across 115 processes on a regular user desktop over one month period).

### 3.1 Windows Registry

Windows Registry is a centralized repository for software, hardware, and user settings on Windows machines. This repository makes it easy for different system components to share and track configurations.

Windows Registry is organized hierarchically, closely resembling a file system. Each Registry entry is uniquely identified by a Registry key and a Registry value. A Registry key resembles a directory and a Registry value a file name. A key may contain multiple subkeys and values. Given a key/value pair, Windows Registry maps it to Registry data, which resembles the content of a file. Hereafter, we will refer to Registry keys, Registry values, and Registry data as Keys, Values, and Data.

Table 1 shows a Windows Registry entry example. Its Key is a hierarchical path name with root Key HKEY_LOCAL_MACHINE, which stores settings generic to all users. The Key in the example stores settings about the Perl application. The Value/Data specifies that the Perl executable is located at C:\Perl\bin\perl.exe. Windows Registry supports about 30 operations (e.g., Createkey and OpenKey), each with a return value indicating the success or failure of the operation. Table 1 shows a successful QueryValue operation (given a Key/Value pair, fetch associated Data).

Previous studies have shown that a significant fraction of configuration errors are due to Windows Registry corruptions [7]. Software bugs, user mistakes, or application updates can all trigger unexpected Registry modifications that lead to software errors. In many cases, even a single entry corruption may result in serious application failures ranging from user-interface changes (e.g., a menu or icon missing) to software crashes.

While Windows Registry facilitates configuration access, it remains challenging to detect and diagnose configuration errors due to the complex and dynamic nature of Windows Registry. The number of Registry entries

| | |
|---|---|
| 1 | Key: HKLM\Software\Policies\Microsoft\Windows\WindowsUpdate<br>Op: OpenKey, Status: success, Value: “”, Data: “” |
| 2 | Key: HKLM\Software\Policies\Microsoft\Windows\WindowsUpdate<br>Op: QueryValue, Status: success, Value: WUServer,<br>Data: http://sup-nam-nlb.redmond.corp.microsoft.com:80 |
| 3 | Key: HKLM\Software\Policies\Microsoft\Windows\WindowsUpdate<br>Op: QueryValue, Status: sucess, Value: WUStatusServer,<br>Data: http://sup-nam-nlb.redmond.corp.microsoft.com:80 |
| 4-26 | ..., ... (check other settings) |
| 27 | Key: HKLM\Software\Policies\Microsoft\Windows\WindowsUpdate\AU<br>Op: QueryValue, Status:not exist,<br>Value: DetectionFrequencyEnabled, Data:“” |
| **28 (normal)** | **Key: HKLM\Software\Policies\Microsoft\Windows\WindowsUpdate\AU**<br>**Op: QueryValue, Status:not exist, Value: NoAutoUpdate, Data:“”** |
| **28 (error)** | **Key: HKLM\Software\Policies\Microsoft\Windows\WindowsUpdate\AU**<br>**Op: QueryValue, Status:0, Value: NoAutoUpdate, Data:1** |
| 29-45 (normal) | ..., ... (check other settings) |

Figure 1: Registry access sequence of Windows update.

is huge—about 200K for an average machine, and this number is increasing [26]. Furthermore, Registry updates are highly frequent. As shown in Table 2, the number of updates can be as high as tens of thousands per process per day. Despite several recent proposals for automatic mis-configuration diagnosis, configuration-error detection remains an open problem.

### 3.2 A Motivating Example

In this example, we illustrates how CODE can detect and diagnose a real-world configuration error that disables the Windows automatic update feature (i.e., switch the OS to the manual update mode).

Given that Windows update often runs as a background task, users who normally leave automatic-update on will hardly notice that their computers have stopped checking for updates. Previous tools do not help in this case because they diagnose configuration errors only after users detect them. Consequently, this error may go undetected, leaving security vulnerabilities not patched and machines compromised. Early detection is thus critical to alert users to reset this important safety feature.

This configuration error was reported when a user removed a program that he or she thought was extraneous [30]. The program removal adds a Value “NoAutoUpdate” and Data 1 under the Key

$K$ = [HKLM\Software\Policies\Microsoft\Windows\WindowsUpdate\AU]

Since an average process can have over 2000 Registry modifications (i.e., writes) per day during its normal execution (Table 2), we need to determine which modifications are relevant to detection. One approach is to monitor and report modifications to only frequently accessed Keys. However, our experiments show that this approach would generate 154 false alarms per desktop/day, an unacceptably high number.

In our detection, CODE identifies that a rule involving a frequent sequence of exactly 45 Registry accesses is violated. By examining this sequence and its occurrence timestamps, we find that these events are issued by an `svchost.exe` process, which synchronizes with an update server and checks for available updates periodically (once per hour for Windows laptops). If there are updates available, the checking process will proceed to download and install the updates.

Figure 1 shows this 45-event sequence. It begins with an OpenKey operation on registry Key “HKLM\...\WindowsUpdate”, which stores all the information about Windows update. Next, `svchost.exe` accesses this Key and all its Values. For example, the second and third operations show that `svchost.exe` queries the URLs of the windows update server and the status reporting server.

At the 28th event, `svchost.exe` queries the Value “NoAutoUpdate” (highlighted in Figure 1). Since this Value does not exist during normal execution, the QueryValue operation will return “Value not found” and `svchost.exe` continues to check other automatic update options. However, after the Value “NoAutoUpdate” is created with Data 1, the operation returns “Success”, causing `svchost.exe` to prematurely stop without further checks.

Since the 45-event sequence occurs frequently in normal execution, CODE will learn a set of rules from this sequence. In particular, it will identify the first 27 events as the context for the 28th event. Thus, in the error case, CODE successfully detects the deviation. In addition, CODE knows (1) the context, the expected event and the event actually happened and (2) what process and the time at which the process created the problematic Value. It can thus pinpoint the root cause and recover the error.

## 4 System Overview

From a high level, our approach identifies *predictable* configuration-access rules from program executions for error detection and diagnosis. From the example described in Section 3, we see that each Windows update check triggers a sequence of 45 Registry accesses. This entire sequence is deterministic and thus predictable. This behavior is not surprising, as configuration-access patterns are usually reflective of a program's control flow. When a program runs the same code blocks with same/similar user inputs, the set of external events tend to be same/similar and in order.

We focus on only these predictable event sequences in our detection. For each event in such a sequence, its preceding event subsequence provides the *context* for the current program execution point. A deviation from the predicable event sequence suggests that the corresponding program's control flow might have changed, which

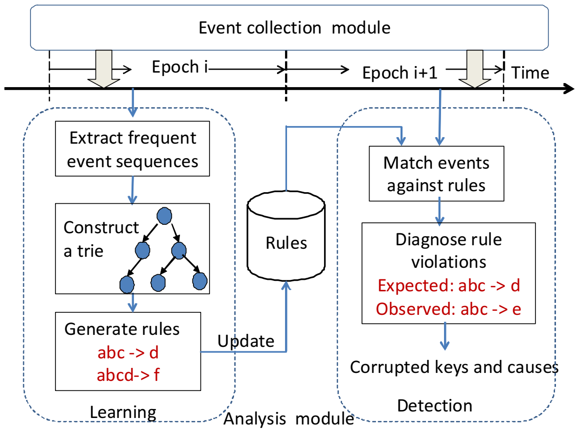



Figure 2: The CODE system architecture.

may indicate the existence of configuration errors. In this case, the expected sequence and the actually observed one are further used to diagnose the error's root cause.

However, not all configuration-access events are predictable. A program's runtime behavior such as caching, optimizations, or the use of temporary files may all affect the program's control flow. Correspondingly, accessing the configuration data will be less predictable. We may observe a large number of temporary events, and even the same set of events may exhibit completely different timing orders. The challenge is how to differentiate the two cases and identify only predictable patterns from a voluminous number of events.

Given the complexity and dynamics of Windows Registry, CODE must meet the following two requirements to realize online detection and diagnosis:

- **Efficient:** The tool should have low timing complexity in order to process events as they arrive in real time. The number of Registry events to process is on the order of $10^6$ to $10^8$ per machine per day.
- **Effective:** As an online tool, CODE needs to distinguish true errors from volatile or benign changes.

We implement CODE as a stand alone tool, monitoring each host independently (Section 8 discusses our deployment of CODE as a centralized manager for data centers). We structure CODE into two parts: an event collection module and an analysis module (Figure 2). Both run simultaneously as a pipeline. The collection module writes Registry operations to disk and the analysis module reads them back for learning, detection, and diagnosis. We chose this architecture to keep the collection module simple; otherwise, it may perturb the monitored processes. We chose files as the communication method between the two modules (instead of sockets) for flexible control over analysis frequency (e.g., every few seconds to minutes).

The core of the event collection module is a Windows kernel module written in C++, similar to FDR [23]. It intercepts all Registry operations and stores them in a buffer in highly compressed forms. It then writes them to disk periodically.[2] Each event contains the following fields: event time-stamp, program name represented by the entire file system path to the executable, command line arguments, process ID, thread ID, Registry Key, Value, Data, operation type (e.g., OpenKey and QueryValue) and operation status.

The analysis module is implemented in C#. It includes a learning component and a detection/diagnosis component. Both learning and detection are done by analyzing the event sequences at a per-thread level because they faithfully follow the program's control flow in execution. For compact representation, we fingerprint a Registry event to generate a Rabin hash [18] by considering all of its fields excluding the time-stamp, the process ID, and the thread ID.

$$\underbrace{a\ b\ c}_{\text{context}} \longrightarrow d$$

Figure 3: Example of a rule.

The learning component takes the Registry event sequences as input, and generates a set of *event transition rules*. Figure 3 shows an example rule. In this example, $a, b,$ and $c$ each represents a unique Registry event. This rule means if we have observed events $a$, $b$, and $c$ in sequence, then the next event is determined to be $d$. In other words, event sequence *abc* is the context of event *d* if and only if *abc* will be always followed by *d* with no exceptions (We do not consider non-deterministic cases where *abc* can be followed by other events such as *e*, as majority of the important errors can be captured by deterministic cases in our experience).

We further require the number of occurrences of the rule sequence to exceed a certain threshold for it to be deemed as a rule. We use the complete command line that launched a process to group the set of threads sharing the common executable name and input arguments[3]. The frequency of a rule is thus measured over all the event sequences across a process group. Finally, the set of learned rules are updated periodically by *epochs* and stored in a rule repository as illustrated in Figure 2. We define an *epoch* as a time period where we observe a fixed number of events, so that the rules learned from one epoch can be applied immediately in the next epoch.

The detection component takes the set of learned rules and applies them to detect errors as new events arrive.

[2]The overhead is negligible, even when flushing the buffer every minute [23].

[3]Different arguments often lead to different program execution paths for different tasks. Applications that launch at machine boot time often start with fixed arguments. In Windows, many applications have a graphical icon on the desktop that launches the application with fixed arguments each time.

In case of a rule violation, the detection module performs a set of checks to facilitate diagnosis based on the rule sequence, the expected event, the Registry write that caused the error, and the actually observed event. In the next two sections, we describe the details of rule learning and error detection/diagnosis.

## 5 Learning Configuration Access Rules

This section describes how CODE generates event transition rules from input Registry-access sequences. These input sequences consist of registry accesses at thread granularity for each process group (i.e., all processes that share the same executable name and command-line arguments). Figure 2 shows the three steps of this procedure: (1) generate frequent event sequences, (2) construct a trie (i.e., a prefix tree) to represent the event transition states, and (3) derive invariant event transition rules based on the trie. For efficient detection, CODE represents the set of output rules in the form of a trie with labeled edges, and each process group has a separate trie.

Throughout the process, CODE has time complexity linear in the number of events processed. Although CODE generates a set of frequent event sequences independently from each epoch, meaning that a sequence has to appear frequently enough within one epoch to be learned by CODE, it maintains the labeled tries in memory across epoches and updates them incrementally. We will show in Section 7.3 that the generated trie sizes are small for most of the programs.

### 5.1 Frequent Sequence Generation

The first step of generating frequent sequences is the most critical, since it provides the candidate event sets for generating rules as well as the potential context lengths. To identify frequent event sequences, one option is to generate hash values for fixed-length event subsequences, and then count their frequencies. We may potentially leverage data structures such as bloom filters [3] to optimize space usage. However, this option is not desirable because it is difficult to pre-determine the event subsequence lengths. Although we may choose several popular lengths (e.g., 2, 4, 8), the semantically meaningful event sequences can be very large (as illustrated in Section 3) and can have varied lengths. Popular techniques such as suffix trees [16] are not applicable either. They typically require the entire input sequence to be available. Furthermore, their space-time requirements are not efficient enough to deal with a large number of Registry events arriving in real time.

In order to generate the longest applicable frequent subsequences efficiently, CODE adopts the Sequitur [17] algorithm. Given a sequence of symbols, Sequitur identifies repeated sequence patterns and generates a set of

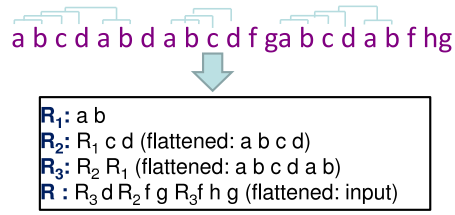



Figure 4: An example of Sequitur hierarchical rules. We also show the flattened rule in the parenthesis.

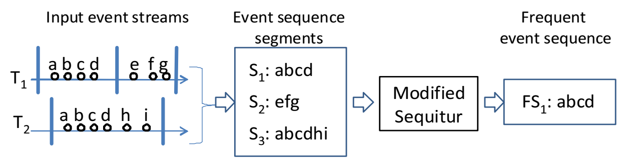



Figure 5: Generating frequent event sequences. $T_1$ and $T_2$ are two threads belonging to the same process group.

grammar rules to hierarchically represent the original sequence. Figure 4 shows an example input sequence and the hierarchical grammar rules derived by Sequitur. The lower case letters represent the input symbols, and we use upper case letters to denote the derived symbols.

During learning, the default epoch size is 500K events, which can span from hours to days for different processes.[4] For each epoch, CODE does not need to store the complete input sequence because the hierarchical representation makes the original sequence more compact. In practice, the number of symbols to store in memory is roughly on the order of the number of distinct Registry events, which is around only 1% of the total events [23].

Compared with other methods, Sequitur has a linear time complexity and reads only one pass of data in streaming mode. Although it may generate sub-optimal frequent sequences, we found it acceptable in our application, as low time complexity is an important requirement. To apply Sequitur in our context, we make the following two modifications to the algorithm:

**Analyzing multiple sequences simultaneously.** The incoming events processed by CODE contain not a single event sequence, but multiple sequences. These sequences come from different processes and different threads in the same process group. In addition, we observe that events belonging to the same task often occur in a bursty manner. Mixing events from these semantically different tasks as one sequence would create unnecessary noise. We thus segment them into per-thread per-burst sequences (the default time interval between two bursts is one second), as shown in Figure 5.

The original Sequitur algorithm, however, analyzes only one sequence at a time. We thus modify it to take multiple sequences. We could maintain a separate gram-

[4] After learning, the detection takes place in real time.

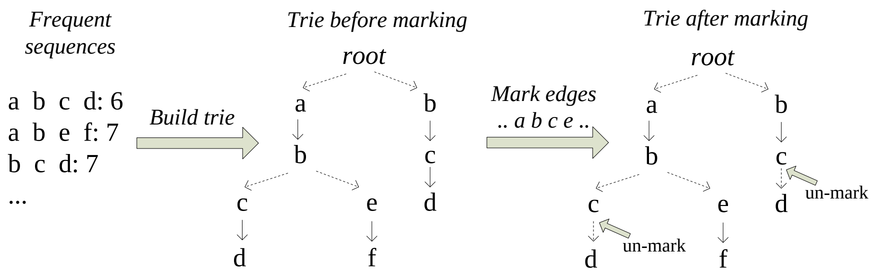



Figure 6: Constructing a trie from frequent event sequences and identifying its rule edges.

mar table (needed for Sequitur) for each sequence, but this approach would miss common subsequences shared across different threads in the same group. For example, in Figure 5, both threads share the subsequence `abcd`. Thus a grammar table is shared among all sequences. This sharing also reduces CODE's memory usage.

With grammar table sharing, one complication arises when a sequence $S_x$ completely contains another one $S_y$. To avoid storing the same sequence twice, Sequitur would replace the redundant copy of $S_y$ in $S_x$ with a pointer to $S_y$. However, we cannot expand $S_y$ if new events come in, because this expansion may make $S_y$ no longer a subsequence of $S_x$. To solve this problem, we give $S_y$ a fresh name $S'_y$ each time we expand it.

**Flattening the hierarchy:** The second modification is to flatten the default hierarchical symbols output by Sequitur to event symbols in order to construct the trie later (illustrated by Figure 4). To ensure each learned sequence is not too short, we select a flattened event sequence only if its length is above a pre-defined length threshold $l$ (by default $l = 4$) and its sequence is above a pre-defined frequency threshold $s$ (by default $s = 5$). We call the frequency of an event sequence as its *support*.

Although the rule flattening process is relatively straightforward, correctly computing the support (i.e., frequency) of the expanded sequences is a more involved task. In Figure 4, $R_1$ appears at both $R_2$ and $R_3$, and $R_2$ further appears at $R_3$. CODE takes a top-down approach to traverse the hierarchical representations for computing the correct support. The final output of this step is a set of frequent event sequences with support greater than $s$.

## 5.2 Event Trie Construction

After CODE generates the frequent sequences from input events, it proceeds to construct an event trie in the form of a prefix tree to store all the frequent sequences from all threads of each process group. Figure 6 shows the construction of an example trie. In a trie, each node represents a Registry access event (encoded as a Rabin hash), and each directed edge represents the transition between the two corresponding events in temporal order.

The adoption of a trie representation serves a couple of important purposes. First, it represents the temporal transition relationships between different events, providing the basis for deriving event transition rules. Second, we found that many frequent event sequences have common prefixes. Hence a prefix tree explicitly encodes the divergence of different event paths from a single point.

We further optimize the trie data structure to make it more compact. An observation is that many event sequences share suffixes as well. In practice, merging common suffixes is very effective in reducing the trie size (by half). Meanwhile, this optimization still preserves the event transition relationship and ensures the correctness of the derived rules.

## 5.3 Rule Derivation

With a trie, CODE proceeds to derive *event transition rules* that all threads from the same process group have to follow. We look at only the rules that were never violated. Our approach is to identify those event transitions *a* → *b* that are deterministic given the sequence of events from the root to *a*. We define such an edge as a *rule edge*. Clearly, only edges from nodes with only one outgoing edge are rule edge candidates.

However, simply counting outgoing edges is incomplete. For example, given a frequent sequence $abcd$, we can construct a trie of 4 nodes, and the edge from $c \rightarrow d$ appears to be a rule edge. However, there may exist a sequence $abce$ that did not occur frequently enough to be selected as a popular sequence. In this case, the transition $c \rightarrow d$ is not deterministic.

For each newly created rule edge, CODE determines whether it is truly a deterministic transition by checking it against the upcoming event sequences in the next epoch. Figure 6 shows this edge-marking process. Doing so defers the use of this edge for detection. It is worth noting that for each event, CODE identifies all possible matches based on the preceding subsequences. Additionally, CODE also starts from the root every time to capture subsequences that begin with the current event. During the edge-marking process in Figure 6, if the incoming event *e* is following sub-sequence *abc*, we will un-mark the two *c* → *d* transitions from rule edge in the trie.

## 6 Error Detection and Diagnosis

This section describes how CODE detects configuration errors using the learned rules and further outputs diagnosis information. Since the labeled trie structure captures the rules as deterministic event transitions and is efficient at matching sequences, we conveniently reuse this data structure for error detection without explicitly representing the rules. The detection algorithm is thus simple and similar to the edge-marking process in Figure 6, except when we see a violation, we report a warning rather than un-marking the transition. This online detection method ensures that we can detect a configuration error as early as possible, before it affects other system states.

### 6.1 False Positive Suppression

In the rule-learning process, the support threshold $s$ can be used to configure the false positive rate. A larger $s$ usually implies a smaller false positive rate, but we may also miss some real errors. We further evaluate this parameter in Section 7.2.

Additionally, we use three techniques to reduce CODE's false positive rate. First, before CODE reports a warning, it performs an additional check to ensure that the violated (i.e., expected) event does not appear in the near future. So if $abc \rightarrow e$ is a rule that is violated by observing $abc$ followed by $f$, then we monitor the events for a delay buffer (set to 1 sec) to check if $e$ appears; if it does, we suppress the warning. The idea behind this check is that since we are looking for corruptions of Registry Keys/Values, if $f$ is indeed a corruption of the Key/Value corresponding to the Registry in event $e$, then $e$ should not appear again. Otherwise it is perhaps simply a benign program flow change.

Second, if multiple alarms are generated in a 1 second delay buffer, CODE only reports the first one as the others are likely manifestations of the same root cause. We found the first alarm is always the true root cause in our experiments (see Section 7.1.1).

The third technique is *cooperative false positive suppression*: aggregate warnings from all machines, and report only unique ones. We consider two warnings identical if they warn about the same Key, Value, and Data. We canonicalized user names when comparing Registry Keys (More canonicalization would help, but it is beyond the scope of this paper). This technique effectively reduced the number of false positives by 30% in our experiments, though it can be turned off for privacy concerns.

### 6.2 Error Diagnosis

CODE also provides rich diagnosis information after error detection. When a process violates a rule $context \rightarrow event$, CODE knows precisely the context, the expected event, the violating event, and the violating process. Such information can help diagnosis in a few ways.

First, CODE allows the operator to understand how the Registry in the expected event was changed by tracking which process, at what time, modified the entry that caused the error. To do so, CODE uses a modification cache to store the last modification operations (along with timestamps) on the Registries in the rules. Because the rules track only frequently accessed Registries and the majority of the accesses to these Registries are read-only events, we need only a small cache. In practice, the size of the modification cache is always smaller than 2,000 events for all the machines that we used in our experiments. The typical size of 200 events is enough for the majority of them.

Second, the expected event and its context often provide enough information regarding the program's anomalous behavior to the administrator. They also provide the candidate Registry entries for recovery. In the "auto-update error" example in Section 3.2, the expected event has empty Data for Value `NoAutoUpdate`, while the violating event has "1" as the Data. Further the expected event belongs to a sequence where svchost.exe is checking for auto-update setting. Such information provides hints to the administrators about the root causes.

Finally, CODE returns all the processes whose rule repositories involve the corrupted Registry. Operators can use this information to examine whether the same configuration error might affect other programs.

## 7 Evaluation

We deployed CODE on 10 actively used user desktops and 8 production servers. In our month-long deployment, we set the data collection interval to every one hour. We ran the analysis module separately off-line on the collected registry-event logs. This allowed us to conveniently examine the logs in detail. For the off-line analysis, it took about 12 hours to process each machine's one-month log. We also evaluated the same version of CODE using one minute intervals to measure its online analysis performance.

To demonstrate the value of using context, we also implemented a *state-based approach* that does not use context for error detection and compared it with CODE. Instead of looking at sequences, this approach tracks commonly used Registry Key/Value entries and raises an alarm if the Data field has not been observed before. To ensure a fair comparison, we applied the same parameters used by CODE as well as the set of false positive suppression heuristics described in Section 6 whenever applicable. Below we present our evaluation results.

### 7.1 Detection Rate and Coverage

We first evaluate CODE using real-world configuration errors and randomly injected Key corruptions.

| Error name | Description |
|---|---|
| Doubleclick | When double clicking any folder in explorer, “Search Result” window pops up. |
| Advanced | IE advanced options missing from menu. |
| IE Search | Search dialog will always be on the left panel of IE that can’t be closed. |
| Brandbitmap | The animated IE logo disappears. |
| Title | IE title changed to some arbitrary strings. |
| Explorer Policy | Windows start menu becomes blank. |
| Shortcut | In explorer, clicking the shortcut to a file no longer works. |
| Password | IE can no longer remember the user’s password. |
| IE Offline | IE would launch in offline mode and user’s homepage can’t be displayed. |
| Outlook trash | Outlook asks to permanently delete items in the “Deleted Items” folder every time it exits. |

Table 3: Description of the 10 reproduced errors.

#### 7.1.1 Detection of Real-Errors

The real world error discovered by CODE was caused by Hotbar Adware [21], which unexpectedly infected one co-author’s desktop. This adware adds graphical skins to Internet Explorer (IE), and modifies a group of Registries related to the Key “HKLM\Software\Classes\Mime\Database\Content type\App”. CODE successfully detected rule violations at the IE start-up time. CODE further provided diagnostic information to help remove the IE tool bars created by the adware.

Additionally, we manually reproduced 20 real-user reported errors to evaluate CODE. These errors were selected from a system-admin support database. The only criteria we used in our selection was whether these errors were triggered by modifications to Windows Registry and were reproducible.[5] The error reproduction process exactly followed the set of user actions that triggered the software failures as described in the failure report. The 20 errors involved nine different programs, including popular ones such as Internet Explorer, Windows Explorer, Outlook, Firefox.

CODE successfully detected all these reproduced errors. Due to space constraints, we do not describe all of them, but list the 10 representative ones in Table 3. To further evaluate the effectiveness of CODE across different environments, we reproduced these 10 errors in 5 different OS environments (one of them was a virtual machine). Not all of these 10 errors can be reproduced on all 5 machines; out of all combinations, we were able to reproduce 41 cases.

Among these 41 cases, CODE detected 40 cases and missed only 1 case (Table 4). Further investigation on the missing case showed CODE had over-fitted the context for that error; that is, the context learned was longer than that observed after the reproduction. We suspect there might exist two different program flows that preceded the access to the corresponding Registry Key, and CODE learned a longer context than what was observed during detection.

[5]Some errors require special hardware setup or specific software versions to reproduce.

| Machine OS and IE version | Server 03 IE 6 | Vista IE 7 | xp-sp2 IE7 | xp-sp3 IE 7 | xp-VM IE 6 |
|---|---|---|---|---|---|
| Doubleclick | 1 (1) | 1 (1) | 1 (3) | 1 (3) | 1 (2) |
| Advanced | 1 (1) | 1 (1) | 1 (2) | 1 (1) | 1 (6) |
| IE Search | 1 (10) | N/A | N/A | N/A | 1 (7) |
| Brandbitmap | N/A | N/A | N/A | N/A | 1 (3) |
| Title | 1 (1) | 1 (1) | 1 (2) | 1 (3) | 1 (3) |
| Explorer Policy | 1 (1) | 1 (2) | 1 (2) | 1 (5) | 1 (2) |
| Shortcut | 1 (1) | 1 (1) | 1 (3) | 1 (1) | 1 (2) |
| Password | N/A | 1 (2) | 1 (1) | 1 (2) | 1 (2) |
| IE Offline | 1 (1) | 1 (1) | 1 (2) | - | 1 (1) |
| Outlook Trash | 1 (2) | 1 (2) | 1 (2) | 1 (2) | N/A |

Table 4: Detection results of reproduced real errors. The first number in each box is the rank of the root cause event, and the second number in the parenthesis is the total number of violations observed in detection. N/A means we couldn’t reproduce that error on that machine, and “-” is the case CODE missed.

Table 4 lists the total number of violations before CODE aggregated the warnings within the one second delay buffer. In all these cases, the root cause event was the first event that occurred. The other violations all happened in a burst right after the first one. By aggregating warnings ( Sect. 6.1), only the first alarm is reported.

Indeed, manual inspection suggests those additional violations are not false positives but are highly correlated to the root cause. For example, the `Outlook Trash` error is triggered by modifying the Data of Key “HKCU\Software\Microsoft\Office\11.0\Outlook\Preferences\Emptytrash” to 1. This error caused an alert window to pop up on each exit of Outlook, asking whether to permanently delete all items in the “Deleted Items” folder. This alert window is related to another Registry Key “\HKCU\Software\Microsoft\Office\11.0\Outlook\Common\Alerts”, whose settings were changed during the error, causing CODE to report additional violations.

Based on the diagnosis information output by CODE, we can easily recover all the reproduced errors by changing the corrupted Registry entries back to the expected ones. However, due to the complex dependencies between today’s system components, we expect automatic recovery to be a challenging topic for future work.

#### 7.1.2 Exhaustive Key Corruption

To evaluate the coverage of CODE's error detection, we manually deleted every Registry Key that is frequently accessed ($\geq$ 2 times) by a process on a virtual machine. Note that this does not imply CODE can detect configuration errors caused by only Registry deletions. Any change to Registries such as modifications or new Key/Value creations, can be detected by CODE so long as a future access to these modified Registries violates a learned rule. For example, the AutoUpdate error in Section 3.2 was caused by modification to a Registry Data.

The process we chose is Internet Explorer (IE), which has both the maximum number of Registries and distinct Registry Key accesses on a typical desktop machine. We ran a program that simulates user browsing activities by periodically launching an IE browser, visiting a Web site, and then closing the browser. After running this program for two hours (for the learning phase), we deleted every Registry Key that IE accessed more than twice during the two hours, one at at time. After each corruption, we ran the program twice that simulates a user's Web visit and let CODE perform detection. We then recovered the corrupted Key before proceeding to the next Key corruption.

| Total Registry accesses | Registry writes | Distinct Keys |
|---|---|---|
| 2,097,642 | 275,549 (13.1%) | 1,247 |
| Frequent Registry Keys ($\geq$ 2 times) | Successfully corrupted Keys | CODE detected corruptions |
| 783 (62.8%) | 387 | 374 (96.6%) |

Table 5: Summary of the Key corruption experiment.

Table 5 summarizes the statistics and the results. Among the 387 successfully corrupted Keys, CODE detected 374 (96.6%) of them. Note not every frequently accessed Key can be corrupted. Among 783 of the frequent Keys, we successfully found and corrupted only 387 of them. The remaining Keys were temporary to the life time of a particular IE instance. Since our experiment periodically launched a new IE instance, those temporary Keys no longer existed at the deletion time.

In total, CODE failed to detect 13 of the corrupted Keys, among which, 12 are Keys or sub-Keys of the following 4 Keys:

- HKEY_LOCAL_MACHINE\software\ classes\rlogin
- HKEY_LOCAL_MACHINE\software\ classes\telnet
- HKEY_LOCAL_MACHINE\software\ classes\tn3270
- HKEY_LOCAL_MACHINE\software\ classes\mailto

These Keys store settings about the dynamically linked libraries for handling four application-layer protocols and they are periodically queried by IE. During the exhaustive Key-corruption experiment, we deleted a Registry Key "AutoProxyTypes" that stores settings about automatic Internet sign-up and proxy detection. The deletion of this Key may have triggered persistent program behavior changes in IE, which switched to an alternative configuration option that did not rely on the above four Keys to perform Internet sign-up and proxy detection. This example also suggests that recovering from errors triggered by configuration changes may require more than reversing these modifications.

| Total Frequent Key Access | Frequent Access CODE Captures | Distinct Accesses To Frequent Keys |
|---|---|---|
| 2,090,777 | 2,083,912 (99.7%) | 2,400 |
| Distinct Access CODE captures | Accesses with single context | Average Number of contexts |
| 2,400 (100.0%) | 1,743,708 (83.4%) | 1.74 |

Table 6: Event context statistics.

To further understand the predictability of using contexts for detection, we measure the number of the Registry accesses that fall into contexts, where our detection is applicable. Table 6 shows that out of the total 2,090,777 accesses to the frequent Keys, 2,083,912 (99.7%) of them fall into some contexts, and thus may be captured by CODE. Furthermore, 83.4% of the frequent accesses belong to a single non-overlapping context. This means that their access happened in only one deterministic way. On average, for each frequent Registry access, it has 1.74 contexts. For those Registry accesses that have more than one context, most of them are related to the settings of dynamically linked modules that may be shared by different components in IE, resulting in more than one context.

### 7.2 False Positive Rate

We evaluated the false positive rate of CODE using month-long Registry access logs from the following two sets of machines: (1) 8 production servers with similar hardware and workloads and (2) 10 desktops used by two interns, four researchers, one research lab manager, and three part-time vendors, giving us a diverse set of workloads. Other than the Hotbar Adware, we were unaware of any other configuration errors reported for the log-collection time period.

| | CODE | | | State-based |
|---|---|---|---|---|
| Num/day/machine | Average | Max | Min | Average |
| Server | 0.06 | 0.27 | 0 | 13.67 |
| Desktop | 0.26 | 0.96 | 0 | 153.83 |

Table 7: Summary of false positive rates (in terms of the number of warnings/machine/day) across 10 desktops and 8 servers.

Table 7 shows the false positive rates of CODE. Over the 30 day period with hundreds of billions of events from all machines, CODE reported a total of 78 warnings with an average of 0.26 warning/desktop/day and 0.06 warning/server/day. As a comparison, the state-based approach reported three orders of magnitude more, on

| Name | Description | Percentage |
|---|---|---|
| File Association | The default program used to open different file types is changed. | 24.1% |
| MRU List | Changes to most recently accessed files tracked by applications (e.g., explorer and IE) | 12.7% |
| IE Cache | The meta-data for the IE Cache entities is changed. | 3.8% |
| Session | The statistics for a user login session are updated. | 3.8% |
| Environment | Environment variable changes. | 2.5% |

Table 8: Top 5 reasons for causing false positives on one machine. The "Percentage" column shows, using the 5 categories, the percentage of alarms that can be summarized over all alarms from all machines.

average 153.83 warnings/desktop/day and 13.67 warnings/server/day. This difference can be explained by several reasons. First, many modifications to frequently accessed Registries do not occur in any frequent sequences (i.e., no context). Second, multiple Registry modifications often belong to a single sequence where CODE reports only the first modification as a warning while the state-based the approach reports all of them. Finally, some modified Registries will never be accessed again after the modification. While the state-based approach reports all such cases as warnings, CODE does not because it reports a warning only when the modified Registry is read again.

We further examine the time distribution of the warnings generate by CODE. Figure 7 shows that for the desktop that generated the largest number of warnings (0.96/machine/day in Table 7), only 4 processes reported a total of 29 warnings during the 740 hours (more than 30 days). Most warnings are clustered in time, and are likely caused by the same configuration modification event.

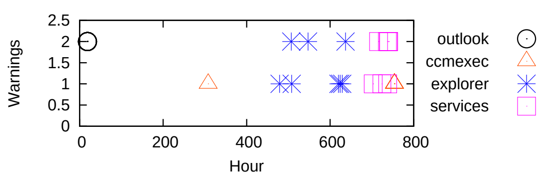



Figure 7: Number of warnings per hour generated by the desktop that had the most number of warnings.

We analyzed the different causes of the false positives on user desktops and found that they can be categorized into a few types (Table 8 summarizes the top five causes). Some of them (File Association and Environment Variable) are intended configuration changes issued by users; the others (Most Recently Used List, IE Cache, and Session Information) are temporary-data changes. By using regular expressions to filter the Registry Keys that fall into these top five causes, we can potentially reduce the false positive rate to $0.14$ warnings/desktop/day.

We also observed a significant overlap in the false positives generated across different machines. Without the cooperative false positive suppression heuristic that merges false positives across machines, the false positive rate in an isolated detection would have increased from $0.26$ to $0.36$ warnings/desktop/day.

#### 7.2.1 Analysis Sensitivity

We study CODE's sensitivity to workload and the support threshold (i.e., the number of occurrences for a frequent event sequence to be learned as a rule) in this section.

**Workload sensitivity.** Table 7 shows that CODE's false positive rate is four times lower on servers than on user desktops. This is because server workloads are less interactive, and thus, their Registry access logs are less noisy. To evaluate the workload sensitivity, we measure the false positive rate of different programs for all the machines in our experiment. Among all the programs running on the servers, only 2 ever reported warnings; for programs running on desktops, 12 reported warnings. The program Windows Explorer (`explorer.exe`) generated the maximum number of warnings, contributing to 1/3 of the total alarms followed by Internet Explorer (`iexplore.exe`) and Windows Login (`winlogon.exe`). Windows Explorer is like the Unix shell for Windows and is highly interactive. While CODE currently uses the same support threshold 5 for learning frequent sequences, we can adjust the false positive rate by setting a larger support threshold.

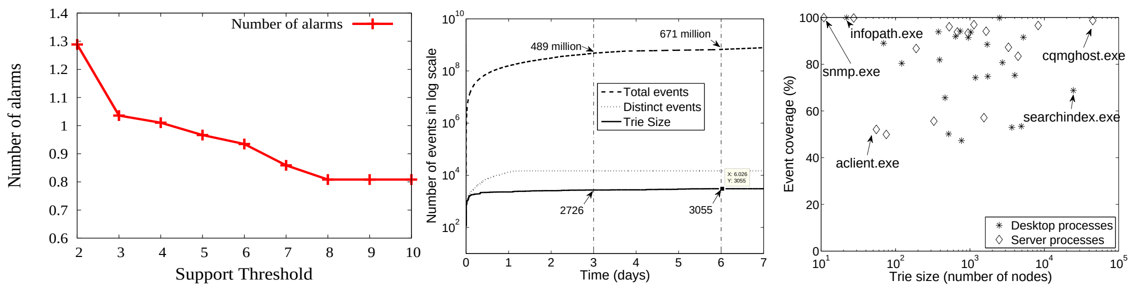
**Support-threshold sensitivity.** As discussed above, an important parameter is the support threshold for separating frequent and infrequent sequences. We evaluated this sensitivity using the desktop with the highest false positive rate (0.96/machine/day in Table 7). Figure 8 shows the result. As was expected, using a larger threshold decreased the false positive rate. Users and administrators can tune this parameter to trade-off detection rate vs. false positive rate.

#### 7.2.2 Impact of Software Updates

Software updates are frequent on modern computers. Their activities may be intrusive and change a program's configuration-access patterns. We study the impact of software updates on the false positive rate in this section.

We used the logs collected from the 10 desktop machines for our analysis. We treat a warning as a software-update related false positive if the corresponding Registry was last modified by one of the Windows software update processes (e.g., `ccmexec.exe`, `svchost.exe`, `update.exe`) and Windows software installation processes (e.g., `msiexec.exe`).

Among the 78 false positives reported by CODE, only 5 were due to software updates, averaging to 0.017 warn-




(a) (b) (c)

Figure 8: (a) Sensitivity of false positive rate vs. support threshold. (b) The growth of the trie size and the number of events over time for IE (desktop) in log scale. (c) Trie size vs. event coverage for different processes on two machines.

ing per desktop/day or 0.139 per update across total 36 updates from these machines. These 5 warnings were caused by two environment variable updates, one display icon update, one DLL update, and one daylight saving start date update. This small false-positive number is not surprising, as software updates tend to fix bugs and add new functionalities, but do not change the existing frequent configuration-access patterns.

CODE learned the new access patterns introduced by software updates as new rules, rather than considering them as false positives. For example, after a large Office update on a desktop, the trie size of the corresponding program increased by 10% within one day. Otherwise, the trie size was relatively stable.

We further examine the most intrusive update we found in the logs: an update from Office Service Pack 2 to Service Pack 3 [5]. This upgrade includes more than 200 patches. It affected 7 of the Office applications, created and modified more than 20,000 keys, but caused only one false positive warning. A closer look revealed that while this update created many keys, the majority of them were temporary keys for bookkeeping and were deleted right after the update, causing no warnings. This update additionally modified or deleted 61 existing keys; only 10 keys overlapped with the rules CODE learned and they were all captured in one rule, causing the only warning. These 10 keys specified the daylight saving start dates of 10 countries and were frequently queried by Outlook [6], resulting in a CODE rule. When the Office update changed these keys, CODE detected a rule violation.

## 7.3 Performance Evaluation

When we deploy CODE in online mode, where it periodically (every minute) processes Registry events arriving in real time, the CPU overhead is very small–less than 1% over 99% of the time, with a peak usage between 10%-25% (on an AMD 2.41GHz due-core machine). The current memory usage is between 500 MB-900 MB.

[6]Outlook queries these keys to determine how to display the calendar items based on the current time zone.

The memory overhead is largely caused by maintaining sets of tries, one for each process group. Figure 8 (b) plots the trie size growth over time in log scale for an IE process. The trie size is about 2000-3000 and converges roughly after 1 day. In contrast, the number of Registry events can be up to hundreds of millions. Even the number of distinct events is one order magnitude larger than the trie size, suggesting CODE is effective in reducing the event complexity.

We proceed to examine the trie sizes for different processes in Figure 8 (c). For the majority of the processes, their trie sizes are consistently small, on the order of hundreds to tens of thousands of events. The total trie size across all processes on a machine is still small, on average 529,500 per user desktop and 97,042 per server. Given each trie node requires around 12 bytes (8 byte Rabin hash + 4 byte pointer), maintaining all the tries requires around 1MB-6MB in the ideal, optimized case. We suspect a large portion of the current memory overhead is caused by both caching the event sequences during the learning phase and the C# overhead. Such overhead can be potentially reduced by using sampled epoches to reduce the learning frequency, and by re-implementing the analysis module in C++.

Figure 8 (c) also shows the percentage of unique events included in the tries defined as *event coverage*. This metric roughly tracks the Registry-access predictability. We found that most of the processes have over 80% of event coverage. In particular, the `snmp.exe` process running on the server is highly predictable, where a trie with 27 unique events can represent 99.77% of all its Registry access events.

One of our goals is to use CODE to monitor server clusters or data center machines for detecting abnormal configuration changes. A typical server cluster consists of machines with similar hardware, software settings, running similar workloads. In this scenario, CODE could offload the analysis task from each server to a small number of centralized management servers.

We run CODE in a centralized mode, constructing a single centralized trie that consists of all the rules from

| | Trie Size (%) | Memory MB (%) |
|---|---|---|
| 1 machine | 98,042 | 503 |
| 2 machines | 119,503 (21.9%) | 510 (1.4%) |
| 4 machines | 134,892 (12.9%) | 560 (9.8%) |
| 8 machines | 139,918 (3.7%) | 600 (7.1%) |

Table 9: The size and memory usage of a centralized trie constructed by analyzing events from multiple machines. The trie size is monitored after 3 days, and the memory usage is the average usage in one day.

multiple machines. Table 9 shows the growth of the trie size and the memory usage as we increase the number of machines to monitor. As we see, the trie size grows by only 3.7% when the number of machines to monitor increases from 4 to 8. This suggests that rules learned from multiple machines can be applied to other similarly configured machines (i.e., with similar hardware, software and workload). For centralized configuration-error detection, the memory overhead is on average about 0.4% per machine for 16GB-memory servers. We leave it as future work to fully generalize the CODE approach to perform centralized data-center management.

## 8 Discussion

**Limitations:** Not all configuration errors can be detected by CODE. By focusing on changes to configuration data and their access patterns, CODE may not detect errors introduced at system or software installation/setup time. To detect these errors, we can extend CODE to process event sequences across machines, so that errors on one machine can be detected by comparing Registry event sequences from another properly installed machine. Previous work [26] has also showed encouraging results by cross referencing static configuration states in a similar way. If a configuration error is caused by an event without any context, CODE cannot detect it either. However, in our evaluation, we have not encountered such errors.

We have evaluated CODE on only Windows Registry, but we believe CODE's underlying techniques can potentially be generalized to other configuration formats, such as Unix's configuration files under /etc/. However, in Unix, different applications manage their own configuration data in their own format, so it might require per-application instrumentation to collect the configuration data access trace.

CODE can be deployed as both a stand-alone tool running on end user's desktops and a centralized management tool used by system administrators to monitor multiple machines in a data-center or a corporate network. We expect CODE to work better in the latter scenario for the following reasons. First, end users might have no clue on how to deal with warnings for filtering false positives. Second, with centralized management, an end user desktop can be spared from the 500-900MB memory overhead (the event collection component still needs to run on end user machines, but it has a negligible overhead [23]). Third, our cooperative false positive suppression feature requires the sharing of canoncalized configuration entries, which is easier to perform in a centralized-management setting.

**Future work:** Our experiments showed that the noise in event logs varied greatly from program to program—after all, these programs have different purposes, workloads, and users. Currently CODE treats all programs uniformly in learning. However, we envision harnessing program-specific knowledge to further improve our detection accuracy and reduce false positives. In particular we may set a higher support threshold for a noisier program. Another possibility is to rank errors based on the importance of the programs affected by these errors. For example, a warning from `system.exe` (the Windows kernel process) may be more important than a warning from `explore.exe`.

In a distributed setting, CODE can collect a much larger, unbiased set of logs to improve the quality of its rules. In particular, for managing server clusters, the homogeneity of the machines may also help reduce CODE's memory overhead and false positive rates (see Section 6, 7.2, and 7.3). One challenge is canonicalization: the rules CODE learns may contain machine-specific information (e.g. machine names, IP addresses, and user names). We manually added user-name canonicalization in CODE. As future work, we plan to develop automatic or semi-automatic techniques to infer more machine-specific configuration data for canonicalization.

## 9 Conclusion

We presented CODE, an online, automatic tool for configuration error detection. Our observation is rather simple: key configuration access events form highly repetitive sequences. These sequences are much more deterministic than each individual event, thus can serve as contexts to predict future events. Based on this observation, CODE uses a context-based analysis to efficiently analyze a massive amount of configuration events. We implemented CODE on Windows and used it to detect Windows Registry errors. Our results showed that CODE could successfully detect real-world configuration errors with a low false positive rate and low runtime overhead.

## Acknowledgments

We thank the anonymous reviewers and our paper shepherd Dilma Da Silva for their valuable feedbacks. We also thank Marcos K. Auguilera for his detailed comments for improving the paper. We thank Professor Yuanyuan Zhou, the UCSD Opera research group and Marti Motoyama for discussion and paper proofreading.

## References


[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering (ICDE)*, pages 3–14, 1995.

[2] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. of ACM*, 13(7):422–426, 1970.

[4] G. Candea. Toward quantifying system manageability. In *Proceedings of the Fourth conference on Hot topics in system dependability (HotDep)*, 2008.

[5] Description of Office 2003 service pack 3. `http://support.microsoft.com/kb/923618`.

[6] X. Ding, H. Huang, Y. Ruan, A. Shaikh, and X. Zhang. Automatic software fault diagnosis by exploiting application signatures. In *Proceedings of the 22nd conference on Large installation system administration conference (LISA)*, pages 23–39, 2008.

[7] A. Ganapathi, Y.-M. Wang, N. Lao, and J.-R. Wen. Why PCs are fragile and what we can do about it: A study of Windows registry problems. In *DSN'04*.

[8] 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 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 101–111, 2007.

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

[10] Windows still No.1 in server OS. http://www.zdnet.com/blog/microsoft/behind-the-idc-data-windows-still-no-1-in-server-operating-systems/5408.

[11] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *Proceedings of IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 157–166, 2008.

[12] E. Kiciman and Y.-M. Wang. Discovering correctness constraints for self-management of system configuration. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC)*, 2004.

[13] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[14] N. Kushman and D. Katabi. Enabling configuration-independent automation by non-expert users. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010.

[15] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. *SIGSOFT Softw. Eng. Notes*, 30:306–315, September 2005.

[16] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, 23(2), 1976.

[17] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 1997.

[18] M. O. Rabin. Fingerprinting by random polynomials. In *Harvard University Report TR-15-81 (1981).*

[19] J. A. Redstone, M. M. Swift, and B. N. Bershad. Using computers to diagnose computer problems. In *Proceedings of the 9th conference on Hot Topics in Operating Systems (HotOS)*, 2003.

[20] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: improving configuration management with operating system causality analysis. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 237–250, 2007.

[21] Symantec hotbar adware information. `http://www.symantec.com/security_response/writeup.jsp?docid=2003-080410-3847-99`.

[22] Top 5 OSes on Oct 09. `http://gs.statcounter.com/#os-ww-monthly-200910-200910-bar`.

[23] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev. Flight data recorder: Monitoring persistent-state interactions to improve systems management. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[24] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–169, 2001.

[25] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and communications security (CCS)*, pages 255–264, 2002.

[26] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[27] Y. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, and C. Yuan. Strider: A black-box, state-based approach to change and configuration management and support. In *Proceedings of the 17th USENIX conference on System administration*, pages 159–172, 2003.

[28] L. R. Welch. Hidden markov models and the Baum-Welch algorithm. *IEEE Info. Theory Society Newsletter*, 2003.

[29] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[30] Windows automatic update disabled. `http://forums.lenovo.com/t5/Windows-XP-and-Vista-discussion/Windows-automatic-update-disabled/td-p/69380`.

[31] Operating system market share. `http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8`.

[32] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pages 375–388, 2006.

[33] 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 International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pages 143–154.