Automated Reasoning and Detection of Specious Configuration in Large Systems with Symbolic Execution

Yigong Hu  
Johns Hopkins University

Gongqi Huang  
Johns Hopkins University

Peng Huang  
Johns Hopkins University

Abstract

Misconfiguration is a major cause of system failures. Prior solutions focus on detecting invalid settings that are introduced by user mistakes. But another type of misconfiguration that continues to haunt production services is specious configuration—settings that are valid but lead to unexpectedly poor performance in production. Such misconfigurations are subtle, so even careful administrators may fail to foresee them.

We propose a tool called Violet to detect specious configuration. We realize the crux of specious configuration is that it causes some slow code path to be executed, but the bad performance effect cannot always be triggered. Violet thus takes a novel approach that uses selective symbolic execution to systematically reason about the performance effect of configuration parameters, their combination effect, and the relationship with input. Violet outputs a performance impact model for the automatic detection of poor configuration settings. We applied Violet on four large systems. To evaluate the effectiveness of Violet, we collect 17 real-world specious configuration cases. Violet detects 15 of them. Violet also identifies 11 unknown specious configurations.

1 Introduction

Software is increasingly customizable. A mature program typically exposes hundreds of parameters for users to control scheduling, caching, etc. With such high customizability, it is difficult to properly configure a system today, even for trained administrators. Indeed, numerous studies and real-world failures have repeatedly shown that misconfiguration is a major cause of production system failures [32, 43, 45, 60].

The severity of the misconfiguration problem has motivated solutions to detect [35, 61, 63], test [37, 57], diagnose [19, 21, 50, 52, 54] and fix [39, 48, 53] misconfiguration. While these efforts help reduce misconfiguration, the problem remains vexing [1–3, 5–10, 17, 18, 31]. They focus on catching invalid settings introduced due to user mistakes. But another type of misconfiguration that haunts production systems, yet not well addressed, is valid but poor configuration. For simplicity, we call them specious configuration.

Specious configuration has a broad scope. In this paper, we focus on—and use the term to refer to—valid settings that lead to extremely poor performance, which is a common manifestation in production incidents. This scope of focus is different from suboptimal configuration (Figure 1). The latter happens when a setting does not yield the best performance, but the performance is still acceptable. This scope is also complementary to efforts on automated configuration performance tuning [33, 51, 62, 64] to search for the best setting.

Take a real-world specious configuration that caused a service outage as an example. An engineer changed the request tracing code from a hard-coded policy (always tracing) to be configurable with a tracing rate parameter. This rate parameter was initially set to 0.0. To retain the same tracing behavior as before, she decided to change the parameter to 1.0. Based on her understanding, this change will turn on the tracing for all message requests that come from internal users. But unfortunately, there was a subtle caveat in the code that caused the actual effect to be turning on tracing for all requests from all users, which quickly overloaded all web servers as well as the backend databases, leading to a catastrophic service outage. Interestingly, before rolling out this specious configuration to production, the change in fact went through a canary phase on a small-scale testing cluster, which unfortunately did not manifest dramatic failure symptoms.

Empirical evidence suggests that specious configuration like the above is prevalent. Yin et al. [60] shows that misconfiguration in the form of legal parameters has similar or higher percentage than illegal parameters. Facebook reports [49] that more than half of the misconfiguration in their high-impact incidents during a three-month period are subtle, “valid” config-
2 Background and Motivation

In this Section, we show a few cases of real-world specious configuration from MySQL to motivate the problem and make the discussion concrete. We analyze how specious configuration affects system performance at the source code level. We choose MySQL because it is representative as a large system with numerous (more than 300) parameters, many of which can be misconfigured by users and lead to bad performance.

2.1 Definition

A program expects its configuration parameters to obey certain rules, e.g., paths exist, the min heap size does not exceed the max size. Invalid configurations violate those rules and usually trigger assertions or errors.

We define specious configuration to be settings that are valid but cause the software to experience bad performance when deployed to production. Admittedly, bad performance is a qualitative criterion. Like prior work, we focus on those issues that cause severe degradation and hurt usability. Ultimately, only users can judge whether the performance slowdown is sub-optimal but tolerable or it is intolerable.

Specious configuration has two classes. One is purely about performance, e.g., buffer size, number of threads. Another class is settings that change the software functionality but the changes also have performance impact. Both classes are important and occur in real-world systems. For the latter class, users might want the enabled functionality and are willing to pay for the performance cost. Thus, whether the setting is specious or not depends on users’ preferences. Our solution addresses both forms. Its focus is to analyze and explain the quantitative performance impact of different settings, so that users can make better functionality-performance trade-offs.

2.2 Case Studies

**autocommit** parameter controls the transaction commit behavior in MySQL. If autocommit is enabled, each SQL statement forms a single transaction, so MySQL will automatically perform a commit. If autocommit is disabled, transactions need to be explicitly committed with COMMIT statements. While autocommit offers convenience (no explicit commit required) and durability benefits, it also has a performance penalty since every single query will be run in a transaction. For some users, this performance implication may not be immediately apparent (especially since it is enabled by default). Even if users are aware of the performance trade-off, they might not know the degree of performance loss, only to realize the degradation is too much after deploying it to production. Indeed, there have been user-reported issues due to this setting [13, 15, 60], and the recommended fix is to disable autocommit, and manually batch and commit multiple queries in one transaction.

To quantify the performance impact, we use sysbench [16] to measure MySQL throughput with autocommit configuration.
(b) Insertion-intensive workload.

Figure 2: MySQL throughput for autocommit under two workloads.

```c
1  int write_row() {  
2    ulong trx_commit_complete();  
3  }  
4  if (autocommit) {  
5    if (flush_at_trx_commit - 1) {  
6      trx_commit_complete();  
7    }  
8    else {  
9      else if (flush_at_trx_commit - 2) {  
10     log_group_write_buf();  
11    }  
12    trx_mark_sql_stat_end();  
13  }  
14  }  
15  void mysql_parse(THD *thd) {
16    if (send_result_to_client(thd) <= 0) {
17      mysql_execute_command(thd);  
18    }  
19  }  
20  free_query(list_root->block());  
21  }  
22  void mysql_execute_command(THD *thd) {  
23    if (send_result_to_client(thd) <= 0) {  
24      mysql_execute_command(thd);  
25    }  
26  }  
27  free_query_cache();  
28  lock_tables(open_and_lock_tables(thd));  
29  break;  
30  case SQLCOM_SELECT:  
31    open_and_lock_tables(thd);  
32    invalidate_query_block_list();  
33  case SQLCOM_LOCK_TABLES:  
34    lock_tables(open_and_lock_tables(thd));  
35    invalidate_query_block_list();  
36    break;  
37  case SQLCOM_UNLOCK_TABLES:  
38    unlock_tables(open_and_lock_tables(thd));  
39    invalidate_query_block_list();  
40    break;  
41  case SQLCOM_LOCK_SCHEMA:  
42    break;  
43  case SQLCOM_UNLOCK_SCHEMA:  
44    break;  
45  case SQLCOM_CREATE_INDEX:  
46    break;  
47  case SQLCOM_DROP_INDEX:  
48    break;  
49  case SQLCOM_CREATE_TEMP_TABLE:  
50    break;  
51  case SQLCOM_DROP_TEMP_TABLE:  
52    break;  
53  case SQLCOM_CREATE VIEW:  
54    break;  
55  case SQLCOM_DROP VIEW:  
56    break;  
57  case SQLCOM_REFRESH:  
58    break;  
59  case SQLCOM_STATISTICS:  
60    break;  
61  case SQLCOM_REFRESH:  
62    break;  
63  case SQLCOM_ALTER_VIEW:  
64    break;  
65  case SQLCOM_DROP VIEW:  
66    break;  
67  case SQLCOM_REFRESH:  
68    break;  
69  case SQLCOM_ALTER_SCHEMA:  
70    break;  
71  case SQLCOM_DROP_SCHEMA:  
72    break;  
73  case SQLCOM_ALTER_DATABASE:  
74    break;  
75  case SQLCOM_REFRESH:  
76    break;  
77  case SQLCOM_ALTER_DATABASE:  
78    break;  
79  case SQLCOM_REFRESH:  
80    break;  
81  case SQLCOM_ALTER_DATABASE:  
82    break;  
83  case SQLCOM_REFRESH:  
84    break;  
85  case SQLCOM_ALTER_DATABASE:  
86    break;  
87  case SQLCOM_REFRESH:  
88    break;  
89  case SQLCOM_ALTER_DATABASE:  
90    break;  
91  case SQLCOM_REFRESH:  
92    break;  
93  case SQLCOM_ALTER_DATABASE:  
94    break;  
95  case SQLCOM_REFRESH:  
96    break;  
97  case SQLCOM_ALTER_DATABASE:  
98    break;  
99  case SQLCOM_REFRESH:  
100  break;  
101  case SQLCOM_ALTER_DATABASE:  
102  break;  
103  case SQLCOM_REFRESH:  
104  break;  
105  case SQLCOM_ALTER_DATABASE:  
106  break;  
107  case SQLCOM_REFRESH:  
108  break;  
109  case SQLCOM_ALTER_DATABASE:  
110  break;  
111  case SQLCOM_REFRESH:  
112  break;  
113  case SQLCOM_ALTER_DATABASE:  
114  break;  
115  case SQLCOM_REFRESH:  
116  break;  
117  case SQLCOM_ALTER_DATABASE:  
118  break;  
119  case SQLCOM_REFRESH:  
120  break;  
121  case SQLCOM_ALTER_DATABASE:  
122  break;  
123  case SQLCOM_REFRESH:  
124  break;  
125  case SQLCOM_ALTER_DATABASE:  
126  break;  
127  case SQLCOM_REFRESH:  
128  break;  
129  case SQLCOM_ALTER_DATABASE:  
130  break;  
131  case SQLCOM_REFRESH:  
132  break;  
133  case SQLCOM_ALTER_DATABASE:  
134  break;  
135  case SQLCOM_REFRESH:  
136  break;  
137  case SQLCOM_ALTER_DATABASE:  
138  break;  
139  case SQLCOM_REFRESH:  
140  break;  
141  case SQLCOM_ALTER_DATABASE:  
142  break;  
143  case SQLCOM_REFRESH:  
144  break;  
145  case SQLCOM_ALTER_DATABASE:  
146  break;  
147  case SQLCOM_REFRESH:  
148  break;  
149  case SQLCOM_ALTER_DATABASE:  
150  break;  
151  case SQLCOM_REFRESH:  
152  break;  
153  case SQLCOM_ALTER_DATABASE:  
154  break;  
155  case SQLCOM_REFRESH:  
156  break;  
157  case SQLCOM_ALTER_DATABASE:  
158  break;  
159  case SQLCOM_REFRESH:  
160  break;  
161  case SQLCOM_ALTER_DATABASE:  
162  break;  
163  case SQLCOM_REFRESH:  
164  break;  
165  case SQLCOM_ALTER_DATABASE:  
166  break;  
167  case SQLCOM_REFRESH:  
168  break;  
169  case SQLCOM_ALTER_DATABASE:  
170  break;  
171  case SQLCOM_REFRESH:  
172  break;  
173  case SQLCOM_ALTER_DATABASE:  
174  break;  
175  case SQLCOM_REFRESH:  
176  break;  
177  case SQLCOM_ALTER_DATABASE:  
178  break;  
179  case SQLCOM_REFRESH:  
180  break;  
181  case SQLCOM_ALTER_DATABASE:  
182  break;  
183  case SQLCOM_REFRESH:  
184  break;  
185  case SQLCOM_ALTER_DATABASE:  
186  break;  
187  case SQLCOM_REFRESH:  
188  break;  
189  case SQLCOM_ALTER_DATABASE:  
190  break;  
191  case SQLCOM_REFRESH:  
192  break;  
193  case SQLCOM_ALTER_DATABASE:  
194  break;  
195  case SQLCOM_REFRESH:  
196  break;  
197  case SQLCOM_ALTER_DATABASE:  
198  break;  
199  case SQLCOM_REFRESH:  
200  break;  
201  case SQLCOM_ALTER_DATABASE:  
202  break;  
203  case SQLCOM_REFRESH:  
204  break;  
205  case SQLCOM_ALTER_DATABASE:  
206  break;  
207  case SQLCOM_REFRESH:  
208  break;  
209  case SQLCOM_ALTER_DATABASE:  
210  break;  
211  case SQLCOM_REFRESH:  
212  break;  
213  case SQLCOM_ALTER_DATABASE:  
214  break;  
215  case SQLCOM_REFRESH:  
216  break;  
217  case SQLCOM_ALTER_DATABASE:  
218  break;  
```

Figure 3: Simplified code snippet from MySQL related to autocommit. The elements with orange-colored background represent configuration variables, and the pink ones represent slow operations.

Figure 4: Code affected by query_cache_wlock_invalidate.

```c
uint64_t log_reserve_and_open(uint len) {
  if (len > log->buf_size / 2) {
    log_buffer_extend((len + 1) * 2);  
    len = log->buf_size;  
    len = log->buf_free + len_upper_limit > log->buf_size;  
    log_buffer_flush_to_disk();  
    goto loop;  
  }  
loop:  
}

Figure 5: Code affected by innodb_log_buffer_size.
```

query cache (Θ), forcing them to open the table and wait (Θ) while the write lock is held. Therefore, the effect is additional synchronization that decreases the system concurrency, which in turn can severely hurt the overall system query throughput.

Similar to autocommit, the performance effect depends on the parameters, execution environment and workloads. Specifically, the bad performance is only manifestable with the combination of MyISAM tables, LOCK TABLES statements and other clients doing select type queries on the locked table.

`innodb_log_buffer_size` determines the size of the buffer for uncommitted transactions. The default value (8M) is usually fine. However if MySQL has transactions with large blob/text fields, the buffer can fill up very quickly and incur performance hit. As shown in Figure 5, the parameter setting has two possible performance impacts: (1) if the length of a new log is larger than half of the buf_size, the system will extend the buffer first by calling log_buffer_extend, which in normal cases mainly involves memory allocation. But if other threads are also extending the buffer, additional synchronization overhead is incurred. If the buffer has pending writes, they will be flushed to disk first; (2) if the buf_size is smaller than the free size plus the length of new log, MySQL will trigger a costly synchronous buffer flush operation.

2.3 Code Patterns

Based on the above and other cases we analyze, we summarize four common patterns on how a specious configuration affects the performance of a system at the source code level:

1. The parameter causes some expensive operation like the fsync system call to be executed.

2. The parameter incurs additional synchronization that itself is not expensive but decreases system concurrency.
3. The parameter directs the execution flow towards a slow path, e.g., not using cached result.
4. The parameter triggers frequent crossings of some threshold that leads to costly operations.

The general characteristic among them is that spurious configuration controls a system’s execution flows—different values cause the program or its libraries to execute different code paths. However, the performance impact is also context-dependent—a spurious configuration is bad only when its value and other relevant factors together direct the system to execute a path that is significantly slower than others.

2.4 Approaches to Detect Spurious Config

To detect spurious configuration, operators rely on experience or manuals, which are neither reliable nor comprehensive. A more rigorous practice is to test the system together with configuration and quantitatively measure the end-to-end performance like throughput. However, if the testing does not have appropriate input or related parameters, the performance issue will not be discovered. Also, because the testing is carried out in a black-box fashion, the approach is experimental. The results are tied to the testing environment, which may not have the same hardware, dependencies or scale as the production. For example, in the incident described in Section 1, that spurious configuration was tested, and the result showed a slight increase of logging traffic to a dependent database. But this increase was deemed small, so it passed the testing.

We argue that while the experimental approach is indispensable, it alone is insufficient to catch spurious configuration. We advocate developing analytical approaches for reasoning about configurations’ performance effect from the system code. The outcome from an analytical approach includes not only a conclusion, but also answers to questions “how the parameter affects what operations get executed?”, “what kind of input will perform poorly/fine?”, “does the effect depend on other parameters?”, etc. In addition, the analysis should enable extrapolation to different contexts, so users can project the outcome with respect to specific workload or environment.

A potential approach is static analysis. Indeed, we can leverage the code patterns in Section 2.3 to detect potential spurious configuration. However, mapping them at concrete code construct level requires substantial domain knowledge. Also, the performance effect involves many complex factors that are difficult to be deduced by pure static analysis.

The observations in Section 2.3 lead us to realize that the crux is some slow path being conditionally executed. Thus, we can transform the problem of detecting spurious configuration to the problem of finding slow execution flow plus deducing the triggering conditions of the slow execution.

3 Overview of Violet

We propose an analytical approach for detecting spurious configuration, and design a tool called Violet. Violet aims to comprehensively reason about the performance effect of system configurations: (1) explore the system without being limited by particular input; (2) analyze the performance effect without being too tied to the execution environment.

Our insight is that the subtle performance effect of a specious parameter is usually reflected in different code paths getting executed, depending on conditions involving the parameter, input and other parameters, and these paths have significant relative performance differences. Based on this insight, Violet uses symbolic execution with assistance of static analysis to thoroughly explore the influence of configuration parameters on program execution paths, identify the conditions leading to each execution, and compare the performance costs along different paths. After these analyses, Violet derives a configuration performance impact model that describes the relationship between the performance effect and related conditions. In this Section, we give an overview of Violet (Figure 6). We describe the design of Violet in Section 4.

3.1 Symbolic Execution to Analyze Performance Effect of Configurations

Background. Symbolic execution [24, 38] is a popular technique that systematically explores a program. Different from testing that exercises a single path of the program with concrete input, symbolic execution explores multiple paths of the program with symbolic input and memorizes the path constraints during its exploration. When a path of interest (e.g., with abort()) is encountered, the execution engine generates an input that satisfies the constraint, which can be used as a test case. Compared to random testing, symbolic execution systematically explores possible program paths while avoiding redundancy. Consider this snippet:

```c
void foo(int n) { if (n > 1000) bar(n); else bazz(n); }
```

Testing may blindly test the program many times with different n, e.g., 1, 10, 20, etc., but they all exercise the same path without triggering the call to bar(). If we use symbolic execution, we can explore the two paths of foo by deriving only two concrete values of n to satisfy the path constraints.

Basic Idea. Configuration is essentially one type of input to a program. The basic idea of Violet is simple—make the parameters symbolic, measure the cost along each execution path explored, and comparatively analyze the costs. The path constraints that the symbolic execution engine memorizes characterize the conditions about whether and when a parameter setting is potentially poor. Take Figure 3 as an example. Violet makes variable autocommit symbolic. Function write_row will fork at line 2. The first path goes into the if branch, with a constraint autocommit == 1. When trx.commit.complete is called in the first path, it encounters another parameter flush_at_trx_commit, which is also made symbolic. Two additional paths are forked within that function. While exploring these paths, Violet records a set of performance cost metrics.
Since the subtle performance effect of specious configuration is often only triggered under specific input, besides configuration parameters, Violet can also make the input symbolic. For the example in Figure 3, the input will determine whether the write_row function will be called or not. Only insert type queries will invoke write_row. This input constraint will be recorded so the analysis later can identify what class of input can trigger the specious configuration.

3.2 Violet Workflow

Figure 6 shows the workflow of Violet. The input to Violet is system code and target configuration. We require source code to identify the program variables corresponding to parameters. In addition, as we discuss later (Section 4.3), Violet uses static analysis to assist the discovery of dependent parameters. To symbolically execute the target system, we leverage a state-of-the-art symbolic execution platform S2E [26] and insert hooks into the system code to make parameters and input symbolic. We design the Violet execution tracer as S2E plugins to record the performance results to a trace during state exploration. The Violet trace analyzer conducts comparative cost analysis, differential critical path analysis, etc.

The output is a configuration performance impact model that describes the relationship among configuration constraints, cost, critical path, and input predicate. Violet further provides a checker to deploy with the software at user sites. The checker consumes the constructed configuration model to continuously detect whether a user-site configuration file or update can potentially lead to poor performance. Upon the detection of potential specious configuration, the Violet checker reports not only the absolute performance result, but also the logical cost and critical path to explain the danger. The checker also outputs a validation test case based on the input predict that provides hints to users about what input can expose the potential performance issue.

4 The Design of Violet

In this Section, we describe the Violet design (Figure 6). We need to address several design challenges. First, configurations have intricate dependencies among themselves and with the input, but making all of them symbolic easily leads to state space explosion. Second, conducting performance analysis in symbolic execution is demanding due to lack of explicit assertion point, mixed costs, overhead, etc. Third, deriving performance model from code requires balance between being generalizable (not too tailored to specific input or environment) and being realistic (reflects costs in real executions).

4.1 Make Config Variable Symbolic

The starting point for Violet is to make parameters symbolic. A naive way is to make the entire configuration file a symbolic blob. While this approach is transparent to the target program, it easily leads to path explosion even at the program initialization stage. An improvement could be only making the configuration value string symbolic during parsing. e.g., make_symbolic(value, 2); but the execution would still spend significant time in the parsing. Also the parameter value range will be limited by the string size, e.g., only explore buf_size from 0 to 99.

We should identify the program variables that store configuration parameters and directly make these variables symbolic. Prior works [56,57] observe that the mature software typically uses uniform interfaces such as an array of struct to store parameters. Thus they annotate these interfaces to extract variable mappings in static analysis. For Violet, we need to additionally identify the parameter type and value constraints defined by the program (e.g., 1 to 10) to restrict the symbolic value. This is because we are only interested in exploring the performance effect of valid values.

Since typically all the config variables are readily accessible after some point during initialization, we take a simple but accurate approach: insert a hook function directly in the source code right after the parsing function and programmatically enumerates these variables and makes them symbolic using their type and other info. In this hook function, we read an external environment variable VIO_SYM_CONFIGS to decide which target parameter(s) to make symbolic.

Take MySQL as an example. Its configuration parameters are represented by a number of Sys_var_* data structures in the code, depending on the parameter’s type. We add a make_symbolic API to these data structures, which uses the type, name, value range information to call the Violet library to make the backing store symbolic. Figure 7 shows an example of the added hook API. Then after MySQL finishes parsing its configurations, we iterate through all configuration variables (Figure 8), which are stored in a global linked list called all_sys_vars. If the parameter is in the target set, we invoke its new make_symbolic API.
4.2 Make Related Config Symbolic

The performance effect of a parameter usually depends on the values of other parameters. Thus, if we only make one parameter symbolic while leaving other parameters concrete, we will only explore incomplete execution paths and potentially miss some problematic combination that leads to bad performance. A straightforward solution is to make all parameters symbolic. Since symbolic execution only forks if a symbolic value is used branch conditions, this approach seems to be feasible. However, the problem with this approach is that most combinations of configuration parameters are unrelated but will be explored during symbolic execution.

Figure 9 illustrates the problem. Suppose we are interested in the performance effect of opty. If we simply make all parameters (optx, opty, optz) symbolic in hope of exploring the combination effect, there will be at least 6 execution paths being explored. But opty is unrelated to optx and optz. The performance impact of opty is only determined by the cost of its branches. For large programs, the target parameter could be used deep in the code. Including unrelated parameters in the symbolic set can cause the symbolic execution to waste significant time or get stuck before reaching the interesting code place to explore the target parameter. The analysis result can also cause confusions. For example, it might suggest only when optx=100 & optz==FILE & opty is true will there be a performance issue and miss detecting specious configuration when opty is true but optx <= 100 or optz != FILE.

Therefore, instead of making all parameters symbolic, we carefully choose the set of parameters to symbolically execute together. In particular, related parameters are usually control dependent on each other. We discover the parameter control dependency with methods described in the following Section.

4.3 Discover Control Dependent Configs

Violet statically analyzes the control dependency relationship of parameters to determine a reduced symbolic parameter set. The static analysis result can significantly help mitigate the path exploration problem during symbolic execution phase.

For a target parameter C, Violet identifies two kinds of related parameters to put in its symbolic set. The enable parameter are those that C is control dependent on. The influenced parameters are those that are control dependent on C. Figure 10 shows an example. For target parameter autocommit, it is used in decide_loging_format and write_row; it has an enabler parameter binlog_format, which decides if autocommit will be activated. autocommit itself influences the performance effect of parameter flush_at_trx_commit. Thus, for autocommit, the set of related parameters to make symbolic together is \{binlog_format,flush_at_trx_commit\}.

Informally, program element Y is control dependent on element X if whether Y’s executed depends on a test at X. More formally, control dependency is captured by postdominator relationship in program Control Flow Graph (CFG). Node \( b \) in the CFG postdominates node a if every path from a to the exit node contains \( b \). Y is control dependent on X if there is a path \( X \rightarrow Z_1 \rightarrow \ldots \rightarrow Z_n \rightarrow Y \) such that \( Y \) postdominates all \( Z_i \) and \( Y \) does not postdominate \( X \). We use postdominator as a building block for our analysis. But our notion of control dependency is broader than the classic definition. For example, if (X) { if (Z1) { if (Z2) { if (Y) { foo(); } } } } \}, the classic definition does not regard X and Y as being control-dependent, because Y does not postdominate Z1 or Z2;
it regards Z2 and \( Y \) as being control-dependent. But for us, all the four parameters are control dependent.

Our analysis is divided into two steps. The first step computes the enabler parameters. Violet builds a call graph of the program. For target parameter \( p \), it locates the usage points of \( p \) and extracts the call chains starting from the entry function to the function \( f \) that encloses a usage point. If any caller \( g \) in the call chain uses some other parameter \( q \), we check if the call site in \( g \) that eventually reaches \( f \) is control dependent on the usage point of parameter \( q \) in \( g \). If so, \( q \) is added to the enabler parameter set of \( p \). Violet identifies enabler parameters within \( f \) through intra-procedural control dependency. Our technical report [34] lists the algorithm.

In the second step, Violet calculates the influenced parameters from the computed enabler parameter sets of all parameters. The related config set is a union of the influenced set and enabler set. We also capture control dependency that involves simple data flow. For example,

\[
\text{void query_cache_init() \{ }
  \text{bool is_disabled() \{ }
    \text{if (query_cache_type == 0) return m_cache_is_disabled; }
    \text{m_cache_is_disabled = TRUE; }
  \} \}
\]

any parameter that is control dependent on the regular variable \( m\text{.cache.is.disabled} \) or return value of \( \text{is.disabled()} \) is also considered to be related to parameter \( \text{query.cache.type} \).

The static analysis result can be inaccurate due to imprecision in the alias analysis, call graph, infeasible path problem, etc. Our general principle is to be conservative and over-approximate the set of related parameters for a target parameter. During symbolic execution, having a few false control dependent parameters does not greatly affect the performance or analysis conclusion and they can manifest through the symbolic execution log if they do cause issues.

### 4.4 Execute Software Symbolically

After the target software is instrumented with the symbolic execution hooks, Violet symbolically executes the software with a concrete configuration file. The hook function reads the \( \text{VI0\_SYM\_CONFIGS} \) environment variable and makes symbolic the program variables corresponding to the specified parameter. In addition, the function parses the control dependency analysis (Section 4.3) result file and makes variables in the related parameter set symbolic as well. Other parameters’ program variables get the concrete values from the configuration file. Besides parameters, Violet can also make program input symbolic to explore its influence on the configuration’s performance impact. This is done through either symbolic arguments (\( \text{sym-args} \)) or identifying the input program variables and inserting \( \text{make_symm_call} \) calls in the code.

### 4.5 Profile Execution Paths

To measure the symbolic parameters’ performance effect, Violet implements a tracer on top of the symbolic execution engine, specifically as a set of plugins on the \( S^2E \) platform.

**Measure Function Call Latency.** We measure function call latency by capturing the call and return signals emitted by \( S^2E \) during symbolic execution. To calculate the latency, a straightforward way is to maintain a stack of call record and pops the top element upon receiving a return signal. This algorithm assumes that the call/return signals are paired and the callee’s return signal comes before the caller’s. But we observe this assumption does not always hold under \( S^2E \). We use a safer method based on return addresses to calculate latency. In particular, the Violet tracer records the \( \text{EIP} \) register value, return address, and timestamp on each call and return signal. The records are stored in two lists. Later, the tracer matches call record list with return record list based on return address fields (Figure 11). The latency for a matched function call is the return record’s timestamp minus the call record’s timestamp. The total latency of each state (execution path) can be obtained from the latency of the root function call.

For multi-threaded programs, function calls from different threads can get mixed up. To address this issue, the Violet tracer stores the current thread id in each profile record and partitions the call and return lists by thread id.

**Re-Construct Call Paths.** The tracer records the function call profile to break down total latency and to enable differential critical path analysis (§4.6). To get the call chain relationship, instead of costly stack frame walk, the tracer uses a simple method with little overhead that just assigns each call record a unique incrementing \( \text{cid} \). Later, the tracer reconstructs the call chain by iterating through all call records in order. If (1) call record \( A \)’s \( \text{cid} \) is larger than call record \( B \)’s \( \text{cid} \), (2) the return address of \( A \) is larger than \( B \)’s \( \text{EIP} \) (the start address of that function), and (3) the difference of the two addresses is smallest among all other pairs (i.e., \( B \)’s start address is closest to the return address in \( A \)), then we assign \( A \)’s \( \text{parentId} \) to be \( B \)’s \( \text{cid} \) and update the current distance.

**Measure Logical Costs.** Besides absolute latency, we also measure a set of logical cost metrics by a similar method of capturing low-level signals from \( S^2E \). In particular, for each execution path, we measure the number of instructions, the number of system calls, the number of file \( \text{I/O} \) calls, the amount of \( \text{I/O} \) traffic, the number of synchronization operations, network calls, etc. These logical costs are useful to surface performance issues other than just long latency. They are also crucial for reducing the test environment’s biases and enabling extrapolation of the result to different settings. For example, if the tracer finds one execution path has a much higher number of \( \text{write} \) \( \text{syscalls} \) compared to other paths whereas...
their latencies are similar. This could be an artifact of the test server having a powerful hard disk or a large buffer cache. But the software might perform poorly in a different environment. The Violet tracer maintains a separate performance profile for each execution path (state) so we can compare the performance effect of different paths. We also need to record the path constraints to identify the parameter combination and the class of input that leads to the execution path. The tracer records the final path constraint when an execution path terminates or it exceeds some user-specified cost threshold.

### 4.6 Analyze State Traces

Once the symbolic execution finishes, the Violet trace analyzer parses the performance traces. It then builds a cost table. Each row represents a state (path) that was explored in symbolic execution. The analyzer does a pair-wise comparison of performance in different rows. If the performance difference ratio exceeds a threshold (default 100%), the analyzer marks that state suspicious. The analyzer compares not only the absolute latency metric but also the collected logical metrics. Even if the latency difference does not exceed the threshold but some logical metric does, the analyzer still marks the state.

Not all pair comparisons are equally meaningful when the symbolic execution explored multiple symbolic variables. To elaborate, assume our target parameter is autocommit, which has a related parameter flush_log. Since both are made symbolic, one state could represent constraint autocommit==0 && flush_log==1 and another state could represent constraint autocommit==1 && flush_log==2. In this case, comparing the costs of these two states is not very meaningful.

The analyzer tries to compare state pairs that are most "similar" first. Determining the similarity of two paths can be challenging. We use a simple approach: in one state’s constraints formula, for each constraint involving a related parameter, if it also appears in the other state’s formula, the similarity count is incremented by one. This method is imprecise as it merely checks the appearances, not constraint equivalence. For our use cases, the inaccuracies are generally acceptable. Besides, the analyzer can compare all pairs first, surface the bad state-pairs, and then we can decide the meaningfulness of the suspicious pairs.

For each pair that has a significant performance difference, the analyzer computes the differential critical path. It first finds the longest common subsequence of the call chain records in the two states. Then it creates a diff trace that stores the common records with performance metrics subtracted, as well as the records that only appear in the slower state. The analyzer finally locates the call record (excluding entry) with the largest differential cost and constructs the critical call path based on the cid and parentid of the call records.

When Violet makes the input symbolic, the path constraints in each state will contain constraints about the input. The analyzer separates the input related constraints as input predicate. This is useful to tell what class of input can expose the potential performance issue for the combination of parameter values that satisfies the configuration constraint in a state. The final output from the Violet analyzer is the configuration performance impact model that consists of the raw cost table (Table 1) with configuration constraints, cost metrics, and input predicate for each state, the state pairs that have significant performance difference, and the differential critical paths.

### 4.7 Continuous Specious Config Checker

Violet provides a standalone checker tool to detect specious configuration. It leverages the configuration performance impact model from the analyzer and validates a concrete user configuration file. The checker tool supports three modes:

1. Some config update introduces performance regression.
2. Some default parameter is poor for users’ specific setup.
3. Code upgrade or workload change make old setting poor.

For scenario 1, the checker references the cost table and locates the state(s) that have configuration constraints satisfying the updated parameter’ old value and the parameter’s new value. If the state pair has significant performance difference, the checker alerts the operators and generates a test case based on the input predicate for operators to confirm the performance regression. For scenario 2, the checker validates if the state that the default value lies in appears in some poor state-pair. If so, it means this default value potentially performs significantly worse than another value. For scenario 3, if the system code changes, Violet rebuilds the cost impact table. The checker then identifies if some state in the new table performs much worse compared to the old cost table. If workload changes, the checker validates if cost table rows that previously satisfy the input predicate perform worse compared to rows that satisfy the input predicate now.

### 5 Scaling Violet to Large Software

In this section, we describe the challenges and our solutions for scaling Violet to large software.

#### 5.1 Choice of Symbolic Execution Engine

We initially build Violet on the KLEE [24] symbolic execution engine because it is widely used and convenient to exper-
iment with. However, while KLEE works well on moderate-sized programs, it cannot handle large programs like MySQL. KLEE models the environment (POSIX runtime and libc) with simplified implementation. Large programs use many libc or system calls that are unimplemented or implemented partially/incorrectly, e.g., `fcntl`, `pread`, and `socket`. KLEE also does not support symbolic execution of multi-threaded programs. We spent several months patching KLEE to fix the environment model and add multi-threading support. When we were finally able to run MySQL with KLEE, it took 40 minutes to just pass initialization even without symbolic data.

We thus decided to switch to the S\(^2\)E platform [26]. S\(^2\)E uses real environment with complete OS and libraries. Executing large software would encounter almost no compatibility issues. In addition, S\(^2\)E uses QEMU and dynamic binary translation to execute a target program. For instructions that access symbolic data, they are interpreted by the embedded KLEE engine; but instructions that access concrete data are directly executed on host CPU. Overall, while the choice of using real environment in symbolic execution in general means slower analysis compared to using simplified models like KLEE, executing concrete instructions on host CPU offsets that slowness and allows S\(^2\)E to achieve significant speed-up. After migrating Violet to S\(^2\)E and with some minor adjustments, we can start MySQL server within one minute.

### 5.2 Handle Complex Input Structure

Since specious configuration is often only triggered by certain input, Violet makes input symbolic besides configuration. For small programs, the input type is typically simple, e.g., an integer, a string, which is easy to be made symbolic. However, large programs’ input can have very complex structure. If we make such complex input symbolic, the program may be stuck in the input parsing code for a long time and the majority of the input generated is invalid. For example, we make input variable `char *packet` (32 bytes) in MySQL symbolic and execute it in S\(^2\)E for 1 hour, which generates several hundred test cases, but none of which is a legal SQL query. Even after adding some additional constraints, the result is similar.

This challenge is not unique to our problem domain. Compiler testing [58] or fuzzing [11] also faces this challenge of how to generate valid input to programs like C compiler or DBMS. We address this problem through a similar practice by introducing workload templates. Instead of having the parser figure out a valid structure, we pre-define a set of input templates that have valid structures. Then we parameterize the templates so that they are not fixed, e.g., the query type, insertion value, the number of queries, etc. In this way, we can make the workload template parameters symbolic.

### 5.3 Reduce Profiling Overhead

Profiling large programs can incur substantial overhead. We build Violet tracer using low-level signals emitted by S\(^2\)E rather than intrusive instrumentation. Nevertheless, symbolic execution is demanding for performance analysis as the program runs much slower compared to native execution. Fortunately, Violet cares about the relative performance between different paths. We can still identify specious configuration if the relative differences roughly match the native execution, which we find is true for most cases. Violet conducts differential analyses to capture performance anomalies. We describe three additional optimizations in Violet tracer.

First, the Violet tracer controls the start and end of its function profiler. This is because if we enable the function profiler at the very beginning, it can be overwhelmed by lots of irrelevant function calls. We add APIs in the tracer and will start the tracer when the target system finishes initialization and stop the tracer when the system enters the shutdown phase.

Second, the tracer avoids guest memory accesses and on-the-fly calculation. Accessing memory in an execution state goes through the emulated MMU in QEMU. Violet tracer only accesses and stores key information (most from registers) about the call/return signals. It defers the record matching, call chain and latency calculation to path termination.

Third, Violet will disable state switching during latency tracking if necessary. Since the function profiler calculates the execution time by subtracting the return signal timestamp from call signal timestamp, if S\(^2\)E switches to execute another state in between, the recorded latency will include the state switching cost. This in general does not cause serious problems because the costs occur in all states and roughly cancels out with our differential analysis. But in rare cases, the switching costs can distort the results. When this happens, Violet will force S\(^2\)E to disable state switching.

### 5.4 Path Explosion and Complex Constraints

A common problem with symbolic execution is path explosion, especially when the symbolic value is used in library or system calls. In addition, some library calls with symbolic data yield complex constraints that make the symbolic execution engine spend a long time in solving the constraints.

Violet leverages a core feature in S\(^2\)E, selective symbolic execution [26], to address this problem. Selective symbolic execution allows transition between concrete and symbolic execution when crossing some execution boundary, e.g., a system call. Violet uses the Strictly-Consistent Unit-Level Execution consistency model, which silently concretizes the symbolic value before entering the boundary and adds the concretized constraint to the symbolic value after exiting the boundary. This consistency model sacrifices completeness but it would not invalidate the analysis result. To improve completeness, we add some relaxation rules in Violet without causing functionality errors: 1) if the library call does not add side effect, such as `strlen/strncpy`, we make the return value symbolic and remove the concretized constraint; 2) if the library call has side effect but does not hurt the functionality, such as `printf`, we directly remove the concretized constraint.
We evaluate Violet on four popular and large (up to 1.2M SLOC) open-source software (Table 2): MySQL, PostgreSQL, Apache, and Squid. Violet can successfully analyze large multi-threaded programs (MySQL and Squid) as well as multi-process (PostgreSQL, Apache) programs.

The manual effort to use Violet on a target system is small, mainly required in two steps: (1) add configuration hooks (Section 4.1); (2) supply input templates (Section 5.2). The other steps in the workflow are automated.

Table 2 shows SLOC of the core hooks we add to the four systems. The hook size varies across systems. MySQL hooks are largest in size mainly because the system defines many (22) configuration types ($sys\_var\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"
We evaluate the 17 cases with testing as well. We use popular benchmark tools such as sysbench and ab. For each case, we set the target parameter and related parameters with concrete values from one of the poor states discovered. We enumerate the standard workloads in the benchmark to test the software with the configurations. Since the absolute performance result are difficult to judge, we use configurations from the good states and collect performance result with them as a baseline. If the performance difference ratio exceeds 100% (the same threshold used by Violet), we consider the case detected. In total, testing detects 10 cases, with a median time of 25 minutes.

Violet is not meant to replace configuration performance testing. In theory, exhaustive testing can expose all cases, but the cost of it is not affordable in practice. Violet systematically explores program states while avoiding the redundancy in exhaustive testing (Section 3.1). Even though in some cases, as shown in Table 4, the Violet analysis time is relatively long, Violet is exploring the performance effects thoroughly, including the combination effect with other parameters and input. Therefore, the performance impact models Violet derives are complete. Once the exploration is done, the outcome can be reused many times while testing needs to be done repeatedly.

Another challenge with testing is to find the baseline for good performance. Our experiment above assumes the existence of good configuration, which users may not have. Violet, in comparison, conducts in-vivo, multi-path analysis, so it naturally has baselines to compare with. The analysis enables Violet to collect deeper logical metrics, which can reveal performance issues that end-to-end metrics may not find.

### 7.4 Exposing Unknown Specious Config

Besides detecting know specious configuration, we evaluate whether Violet can expose unknown specious configuration. We first apply Violet to derive performance models for all parameters if possible (Section 7.6). We then analyze the results for parameters not in the known case dataset (Section 7.2).

We manually check (1) if some parameter's default or suggested value is in a poor state; (2) if a poor state of a parameter contains related parameters that are undocumented. The manual inspection involves checking the Violet output, the descriptions in the official documentation and tuning guide.
and running tests to confirm, which takes significant time. We only carefully inspect a subset of the results.

The four systems are very mature and maintain high-quality documentations, so it is not easy to find many errors in them. Indeed, a significant portion of the poor states we examined turns out to be already documented. Still we have identified 11 parameters that have potential bad performance effect and the documentation is incomplete or incorrect.

Table 5 lists the cases. For example, our analysis of `vacuum_cost_delay` shows that a higher value can incur large cost for write-intensive workloads, but the default value is 20 ms. Interestingly, we find PostgreSQL 12 (our experiments use v11) changes the default to 2 ms. For `log_statement`, Violet discovers multiple poor states that are not mentioned in the official document. Our analysis reveals that setting it to `nod` causes performance issues for write query when `synchronous_commit` is off. Violet finds some unexpected parameter combination that leads to bad performance, e.g., `parallel_leader_participation` and `random_page_cost`.

We reported our findings to the developers. Eight reports and running tests to confirm, which takes significant time. We only carefully inspect a subset of the results.

The four systems are very mature and maintain high-quality documentations, so it is not easy to find many errors in them. Indeed, a significant portion of the poor states we examined turns out to be already documented. Still we have identified 11 parameters that have potential bad performance effect and the documentation is incomplete or incorrect.

Table 5 lists the cases. For example, our analysis of `vacuum_cost_delay` shows that a higher value can incur large cost for write-intensive workloads, but the default value is 20 ms. Interestingly, we find PostgreSQL 12 (our experiments use v11) changes the default to 2 ms. For `log_statement`, Violet discovers multiple poor states that are not mentioned in the official document. Our analysis reveals that setting it to `nod` causes performance issues for write query when `synchronous_commit` is off. Violet finds some unexpected parameter combination that leads to bad performance, e.g., `parallel_leader_participation` and `random_page_cost`.

We reported our findings to the developers. Eight reports and running tests to confirm, which takes significant time. We only carefully inspect a subset of the results.

The participants are randomly assigned into two groups: group A (w/ Violet checker help) and group B (w/o checker help). Users in group B can run any tools to help them make the decision. We also tell group A users that they do not have to trust the checker output and are free to run other tools.

Figure 12 shows the accuracy of user study result for each group. Overall, programmers w/ Violet checker’s help have 30% misjudgment rate while programmers with Violet checker’s help only have 5% misjudging rate. Figure 13 shows the time for making a judgment. On average, participants took 20.7% less time (9.6 min. versus 12.1 min.) to make a judgment when they were provided with Violet checker. The reason that time saving is not very large is partly because we explicitly tell users the set of parameters, which creates a biased advantage to group B users; and some of our group A users are extra cautious and spend time running other tools.

### 7.5 User Study on Violet Checker

To understand whether Violet checker helps users catch specious configuration, we conduct a controlled user study with 20 programmers (no authors are included). Fourteen are undergraduate students who have taken the database class. Six are graduate students. They all have decent experience with databases and Unix tools. We further give a tutorial of MySQL and PostgreSQL, the descriptions of the common configuration, and available benchmark tools they can use.

We use 6 target parameters from MySQL and PostgreSQL. For each parameter, we prepare two versions of configuration files. In one version (bad), the parameter is set with the poor value and the related parameters are also set appropriately that would create bad performance impact under a workload. In another version (good), we set the target parameter to a good value, or we change the related parameter values, or we tell users the production workloads are limited to certain types (e.g., read-intensive). So in total, we have 12 cases.

Each participant is given 6 configuration files. They need to make a judgment regarding whether the configuration file would create potential performance issue. Since a configuration file contains many parameters, we explicitly tell users the set of parameters they can focus on, which disadvantages Violet because users in practice do not have this luxury.

We conduct a coverage test of Violet by applying Violet on the four software and try to derive performance models for as many parameters as possible. We manually filter the parame-
we observe some false positives in the Violet performance. These false positives are relatively easy to suppress by absorbing the performance noises in symbolic execution. But the ratio of setting 1 to setting 0 is similar: 1.92 for Violet, 1.94 for S²E, and 1.71 for native execution.

7.8 False Positives

The Violet differential performance analysis in general can absorb the performance noises in symbolic execution. But we observe some false positives in the Violet performance analysis output. For example, S²E somehow has a delay in emitting the return signal of some system call functions like `gettimeofday`, which causes Violet to record inaccurate latency. These false positives are relatively easy to suppress by discounting the cost of the noisy instructions.

We manually inspect the performance models of 10 random parameters that Violet analyzes in the coverage experiment.

7.7 Accuracy of Violet Profiling

Since symbolic execution can introduce significant overhead, it seems that the latency traced by the symbolic engine will not be accurate. However, we observe that while the absolute latency under symbolic execution is indeed much larger than native execution, the comparative results between different paths are usually similar. We add a micro-benchmark experiment to test the latency measurement from Violet, vanilla S²E and native mode. Table 7 shows the result from four representative parameters. Take parA as an example. The latency results from Violet and S²E are much later than native result. But the ratio of setting 1 to setting 0 is similar: 1.92 for Violet, 1.94 for S²E, and 1.71 for native execution.

8 Limitations

Violet has several limitations that we plan to address in future work. First, Violet explores the configuration under normal conditions. Some specious configuration may be only used in error handling. Exploring their effect requires specific faults. One solution is to combine symbolic execution with fault injection. Another potential solution is to use under-constrained symbolic execution [46]. Second, our handling of floating point type parameters is imperfect due to limited support in existing symbolic execution engines. We currently explores float parameters by choosing from a set of concrete floating-point values in the valid value range. Third, we use concrete (the host) hardware in the symbolic execution, which may not
capture specious configuration that is only visible in specific hardware. We rely on logical cost metrics to surface such issues. Lastly, Violet does not work on distributed systems.

9 Related Work

Misconfiguration detection and diagnosis. A wide body of work has been done to detect and troubleshoot misconfiguration [20–22, 27, 30, 30, 48, 50, 52, 54, 61, 63]. For example, ConfAid [21] uses dynamic taint tracking to locate configuration errors that lead to failures; Strider [52] and PeerPressure [50] take statistical approaches to identify misconfiguration; EnCore [63] enhances statistical learning with environment information to detect misconfiguration.

These solutions mainly target illegal configuration and have limited effects on specious configuration. X-ray [19] targets performance-related misconfiguration. Our work is inspired by X-ray and is complementary to it. X-ray is a diagnosis tool and uses deterministic record and replay of a specific program execution. Violet focuses on detecting specious configuration beforehand. Violet uses symbolic execution to explore the performance effect in multiple execution paths. Violet is more suitable for performance tuning/bug finding, whereas X-ray is better at diagnosing misconfiguration that has occurred.

LearnConf [41] is recently proposed to detect performance misconfiguration using static analysis. LearnConf summarizes common code patterns of performance configuration and uses simple formulas to approximate the performance effect, e.g., linear relationship. It uses static analysis to identify these patterns and derive parameters to the formulas. The solution is simpler compared to Violet, but its completeness is limited because obtaining comprehensive code patterns is hard. Moreover, the performance effect is often quite complex, which cannot be accurately captured by simple formulas. Static analysis also suffers from well-known inaccuracies for large software. Violet explores a configuration’s influence in the code without requiring or being limited by common patterns; it analyzes the performance effect by executing the code. Additionally, Violet explores the performance impact of input and a large set of related configurations together.

Performance tuning of configuration. There is a wealth of literature on automatic performance tuning, e.g., [33, 44, 51, 55, 59, 62, 64]. They work basically by devising an approximate function between configuration values and the performance metrics measured through testing. While tunable parameters are common specious configuration, performance tuning and detecting specious configuration are two directions. The former searches for settings that yield the best performance, while the latter identifies settings that lead to extremely poor performance. Violet takes an analytical approach to derive configuration performance impact model from the code, instead of exhaustive testing. The result from our in-vivo, multi-path analysis is also less susceptible to noises and enables extrapolation to different contexts.

System resilience to misconfiguration. ConfErr [37] uses a human error model to inject misconfiguration. SPEX [57] uses static analysis to extract configuration constraints and generates misconfiguration by violating these constraints. The injected misconfigurations are illegal values that can trigger explicit errors like crash. Specious configuration typically does not cause explicit errors.

Configuration languages. Better configuration languages can help avoid misconfiguration. Several works make such efforts [23, 25, 28, 29, 35, 42, 47]. PRESTO [29] proposes a template language to generate device-native configuration. ConfValley [35], proposes a declarative validation language for generic software configuration. These new designs do not prevent specious configuration from being introduced.

Symbolic execution in performance analysis. Symbolic execution [24, 38] is typically used for finding functional bugs. SÆ [26] is the first to explore performance analysis in symbolic execution as one use case to demonstrate the generality of its platform. The Violet tracer leverages the advances made by SÆ, particularly its low-level signals, to build our custom profiling methods (Section 4.5). Our tracer also addresses several unique challenges to reduce the performance analysis overhead (Section 5.3). Bolt [36] extracts performance contracts of Network Function code with symbolic execution. Violet targets general-purpose software and analyzes performance effect of system configuration.

10 Conclusion

Specious configuration is a common and challenging problem for production systems. We propose an analytical approach to tackle this problem and present a toolchain called Violet. Violet uses symbolic execution and program analysis to systematically reason about the performance effect of configuration from code. The derived configuration performance impact model is used for subsequent detections of specious configuration. We successfully apply Violet on four large system software and detect 15 out of 17 real-world specious configuration cases. Violet exposes 11 unknown specious configuration, 8 of which are confirmed by developers.

Acknowledgments

We would like to thank our shepherd, Jason Flinn, and the anonymous OSDI reviewers for their thoughtful comments. We appreciate the discussion and suggestions from Xi Wang. We thank Varun Radhakrishnan and Justin Shafer for their contributions to the Violet tool and study cases. We thank our user-study participants and the open-source developers who responded to our requests. We also thank the SÆ authors, especially Vitaly Chipounov for maintaining the SÆ platform and answering our questions. We thank Chunqiang Tang for the prior collaboration that provided early motivation for this work. This work is supported by the NSF CRII grant CNS-1755737 and partly by NSF grant CNS-1910133.
References


