Early Detection of Configuration Errors to Reduce Failure Damage

Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu*, Long Jin, Shankar Pasapathy†
University of California, San Diego  *University of Chicago  †NetApp, Inc.

Abstract
Early detection is the key to minimizing failure damage induced by configuration errors, especially those errors in configurations that control failure handling and fault tolerance. Since such configurations are not needed for initialization, many systems do not check their settings early (e.g., at startup time). Consequently, the errors become latent until their manifestations cause severe damage, such as breaking the failure handling. Such latent errors are likely to escape from sysadmins’ observation and testing, and be deployed to production at scale.

Our study shows that many of today’s mature, widely-used software systems are subject to latent configuration errors (referred to as LC errors) in their critically important configurations—those related to the system’s reliability, availability, and serviceability. One root cause is that many (14.0%–93.2%) of these configurations do not have any special code for checking the correctness of their settings at the system’s initialization time.

To help software systems detect LC errors early, we present a tool named PCHECK that analyzes the source code and automatically generates configuration checking code (called checkers). The checkers emulate the late execution that uses configuration values, and detect LC errors if the error manifestations are captured during the emulated execution. Our results show that PCHECK can help systems detect 75+% of real-world LC errors at the initialization phase, including 37 new LC errors that have not been exposed before. Compared with existing detection tools, it can detect 31% more LC errors.

1 Introduction
1.1 Motivation
 Failures are a fact of life in today’s large-scale, rapidly-changing systems in cloud and data centers [7, 24, 30, 58]. To mitigate the impact of failures, tolerance and recovery mechanisms have been widely adopted, such as employing data and node redundancy, as well as supporting fast rebooting and rollback. While these mechanisms are successful in handling individual machine failures (e.g., hardware faults and memory bugs), they are less effective in handling configuration errors [20, 24, 28], especially the errors in configurations that control the failure handling itself. For example, an erroneous fail-over configuration resulted in a 2.5 hour outage of Google App Engine in 2010, affecting millions of end users [44]. Moreover, very often, the same configuration error is deployed onto thousands of nodes and resides in persistent files on each node, making it hard to tolerate by redundancy or server rebooting. As a result, configuration errors have become one of the major causes of failures in large-scale cloud and Internet systems, as reported by many system vendors [21, 34, 55] and service providers [7, 24, 28, 43].

Since it is hard to completely avoid configuration errors (after all, everyone makes mistakes; as do system administrators), similar to fatal diseases like cancer, a more practical approach is to detect such errors as early as possible in order to minimize their failure damage:

- Early detection before configuration roll-out can prevent the same error from being replicated to thousands of nodes, especially in the data-center environment.
- Unlike software bugs, configuration errors, once detected, can be fixed by sysadmins themselves with no need to go through developers. Therefore, if detected earlier, the errors can be corrected immediately before the configurations are put online for production.
- For many configurations that control the system’s failure handling, early detection of errors in their settings can prevent the system from entering an unrecoverable state (before any failures happen). Often, the combination of multiple errors (e.g., a configuration error plus a software bug) can bring down the entire service, as shown in many newsworthy outages [9, 42, 43, 45].

Unlike software bugs that typically go through various kinds of testing before releases (such as unit testing, regression testing, stress testing, system testing, etc.), system administrators often do not perform extensive testing on configurations before rolling them out to other nodes and putting the systems online [25]. Besides the lack of skills [25] and the temptation of convenience [24], the more fundamental reason is that system administrators do not have the same level of understanding on how and when the system uses each configuration value internally. Thus, they are limited to simple black-box testing such as starting the system and applying a few small workloads to see how the system behaves. Due to time and knowledge limitations, system administrators typically do not
perform a comprehensive suite of test cases against configuration settings, especially for those hard-to-test ones (e.g., failure/error-handling related configurations) that may require complex setups and even fault injections.

Therefore, early detection should inevitably fall onto the shoulder of the system itself—the system should automatically check as many configurations as possible at its early stages (the startup time). Unfortunately, many of today’s systems either skip the checking or only check configurations right before the configuration values are used, as shown in our study (§2). Typically, at the startup time, only those configuration parameters needed for initialization are checked (or directly used), while many other parameters’ checking is delayed much later until they are used in special tasks. Since such configuration parameters are neither used nor checked during normal operations, errors in their settings go undetected until their late manifestation, e.g., under circumstances like error handling and fail-over. For simplicity, we refer to such errors as latent configuration (LC) errors.

LC errors can result in severe failures, as they are often associated with configurations used to control critical situations such as fail-over [44], error handling [42], backup [37], load balancing [9], mirroring [45], etc. As explained above, their detection or exposure is often too late to limit the failure damage. Take a real-world case as an example (c.f., §2: Figure 3a), an LC error in the fail-over configuration settings is detected only when the system encounters a failure (e.g., due to hardware faults or software bugs) and tries to fail-over to another component. In this case, the fail-over attempt also fails, making the entire system unavailable to all the clients.

Tables 1 and 2 compare the severity level and diagnosis time of real-world configuration issues caused by LC errors versus non-latent configuration errors (detected at the system’s startup time) of COMP-A, a major storage company in the US. Although there have been fewer LC errors than non-latent ones, LC errors contribute to 75% of the high-severity issues and take much longer to diagnose, indicating their high impact and damage.

Table 1: Severity of latent versus non-latent errors among customers’ configuration issues of COMP-A. LC errors contribute to 75% of the high-severity configuration issues.

<table>
<thead>
<tr>
<th>Severity level</th>
<th>Latent</th>
<th>Non-latent</th>
</tr>
</thead>
<tbody>
<tr>
<td>All cases</td>
<td>47.6%</td>
<td>52.4%</td>
</tr>
<tr>
<td>High severity</td>
<td>75.0%</td>
<td>25.0%</td>
</tr>
</tbody>
</table>

Table 2: Diagnosis time of latent versus non-latent errors among customers’ configuration issues of COMP-A. The time is normalized by the average time of all the reported issues.

<table>
<thead>
<tr>
<th>Error class</th>
<th>Mean</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latent</td>
<td>1.14</td>
<td>1.70</td>
</tr>
<tr>
<td>Non-latent</td>
<td>0.87</td>
<td>0.41</td>
</tr>
</tbody>
</table>

Figure 1: A real-world LC error from Squid [37]. The error caused system hanging for 7+ hours, and resulted in 48 hours of diagnosis efforts. Later, a patch was added to check the existence of the configured path during initialization. Unfortunately, the patched check is still subject to LC errors such as incorrect file types and permissions.

Figure 2: A real-world LC error from MapReduce [12]. When the exception handler caught the runtime exception induced by the LC error, it was already too late to avoid the downtime. After this incident, the user requested to check the configuration “at the initialization time.”

Figure 1 shows a real-world LC error from Squid, a widely used open-source Web proxy server. The LC error resided in diskd_program, a configuration parameter used only during log rotation. Squid did not check the configuration during initialization; thus, this error was exposed much later after days of execution. It caused 7+ hours of system downtime and cost 48 hours of diagnosis efforts. After the error was finally discerned, the Squid developers added a patch to proactively check the setting at system startup time to prevent such latent failures.

Figure 2 shows another real-world example in which an LC error failed a large-scale MapReduce job processing. This LC error was replicated to multiple nodes and crashed the TaskTrackers on those nodes. Specifically, the error caused a runtime exception on each node. The TaskTracker caught the exception and restarted the job. Unfortunately, as the error is persistent in the configuration file, restarting the job failed to get rid of the error but induced infinite loops. Note that when the exception handler caught the error, it was already too late to avoid downtime (the best choice is to terminate the jobs).

Preventing above LC-error issues would require software systems to check configurations early during the initialization time, even though the configuration values are only needed in much later execution or during special
circumstances. This is indeed demonstrated by the developers’ postmortem patches. As revealed in Facebook’s recent study [43], 42% of the configuration errors that caused high-impact incidents are “obvious” errors (e.g., typos), indicating the limitations of code review and system testing in preventing LC errors. These errors might be detected by early checks (only if developers are willing to and remember to write the checking code).

### 1.2 State of the Art

Most prior work on handling configuration errors focuses on troubleshooting and diagnosis [5, 6, 32, 46, 48, 49, 52, 53, 56, 60, 61, 62]. The techniques proposed in these work are helpful for system administrators to identify the failure root causes faster to shorten the repair time. However, they cannot prevent failures and downtime.

Most of the existing detection tools check configuration settings against apriori correctness rules (known as constraints). However, as large software systems usually have hundreds to thousands of configuration parameters, it is time-consuming and error-prone to ask developers to manually specify every single constraint, not to mention that constraints change with software evolution [61].

So far, only a few automatic configuration-error detection tools have been proposed. Most of them detect errors by learning the “normal” values from large collections of configuration settings in the field [29, 36, 57, 59]. While these techniques are effective in certain scenarios, they have the following limitations, especially when being applied to cloud and data centers.

First, most of these works require a large collection of independent configuration settings from hundreds of machines. This is a rather strong requirement, as most cloud and data centers typically propagate the same configurations from one node to all the other nodes. Thereby, the settings from these nodes are not independent, and thus not useful for “learning”. Second, they do not work well with configurations that are inherently different from one system to another (e.g., domain names, file paths, IP addresses) or incorrect settings that fall in normal ranges. They also cannot differentiate customized settings from erroneous ones. Furthermore, most of these tools target on specific error types (encoded by their predefined constraint templates) and are hard to generalize to detect other types of errors. A recent work learns constraints from KB (Knowledge Base) articles [31]. However, this approach has the same limitations discussed above. Specially, KB articles are mainly served for postmortem diagnosis and thus may not cover every single constraint.

There are very few configuration-error detection approaches that do not rely on constraints specified manually by developers or learned from large collections of independent settings (or KB articles). The only exception (to the best of our knowledge) is conf spellchecker [35] which detects simple errors based on type inference from source code. While this technique is very practical, it is limited in the types of configuration errors that can be detected, as shown in our experimental evaluation (§4).

### 1.3 Our Contributions

This paper makes two main contributions. First, to understand the root causes and characteristics of latent configuration (LC) errors, we study the practices of configuration checking in six mature, widely-deployed software systems (HDFS, YARN, HBase, Apache, MySQL, and Squid). Our study reveals: (1) In today’s software systems, many (14.0%–93.2%) of the critically important configuration parameters (those related to the system’s reliability, availability, and serviceability) do not have any special code for checking the correctness of their settings. Instead, the correctness is verified (implicitly) when the configuration values are being actually used in operations such as a file open call. (2) Many (12.0%–38.6%) of these configuration parameters are not used at all during system initialization. (3) Resulting from (1) and (2), 4.7%–38.6% of these critically important configuration parameters do not have any early checks and are thereby subject to LC errors that can cause severe impact on the system’s dependability.

Second, to help systems detect LC errors early, we present a tool named PCHECK that analyzes the source code and automatically generates configuration checking code (called checkers) to validate the system’s configuration settings at the initialization phase. PCHECK takes a unique and intuitive method to check each configuration setting—emulating the late execution that uses the configuration value; meanwhile capturing any anomalies exposed during the execution as the evidence of configuration errors. PCHECK does not require developers to manually implement checking logic, nor rely on learning a large volume of configuration data. The checkers generated by PCHECK are generic: they are not limited to any specific, predefined rule patterns, but are derived from how the program uses the parameters.

PCHECK shows that it is feasible to accurately and safely emulate late execution that uses configurations. It statically extracts the instructions that transform, propagate, and use the configuration values from the system program. To execute these instructions, PCHECK makes a best effort to produce the necessary execution context (values of dependent variables) that can be determined statically. PCHECK also “sandboxes” the emulated execution by instruction rewriting to prevent side effects on the running system or its environment.

More importantly, emulating the execution can expose many configuration errors as runtime anomalies (e.g., ex-
crections and error code) and the emulated execution runs in a short period. PCHECK inserts instructions to capture the anomalies that may occur during the emulated execution, as the evidence to report configuration errors.

As an enforcement, PCHECK encapsulates the emulated execution and error capturing code into checkers for every configuration parameter, and invokes the checkers at the system’s initialization phase. This can minimize potential LC errors, and compensate for the missing and incomplete configuration checks in real-world systems.

We implement PCHECK for C and Java programs on top of the LLVM [4] and Soot [3] compiler frameworks. We apply PCHECK to 58 real-world LC errors of various error types occurred in widely-used systems (each leads to severe failure damage), including 37 new LC errors that have not been exposed before. Our results show that PCHECK can detect 75+ % of these real-world LC errors at the system’s startup time. Compared with the existing detection tools, it can detect 31% more LC errors.

2 Understanding Root Causes of Latent Configuration Errors

To understand the root causes and characteristics of LC errors, we study the practices of the configuration checking and error detection in six mature, widely-deployed open-source software systems (c.f., Table 3). They cover multiple functionalities and languages, and include both single-machine and distributed systems.

We focus on configuration parameters used in components related to the system’s Reliability, Availability, and Serviceability (known as RAS for short [50]). For each system considered, we select all the configuration parameters of RAS-related features based on the software’s official documents, including error handling, fail-over, data backup, recovery, error logging and notification, etc. The last column of Table 3 shows the number of the studied RAS parameters. Compared with configurations of other system components, configurations used by RAS components are more likely to be subject to LC errors due to their inherently latent nature; moreover, the impact of errors in RAS configurations is usually more severe.

Note: LC errors are not limited to RAS components. Thus, the reported numbers may not represent the overall statistics of all the LC errors in the studied systems. In addition, PCHECK, the tool presented in §3, applies to all the configuration parameters; it does not require manual efforts to select out RAS parameters.

2.1 Methodology

We manually inspect the source code related to RAS configuration parameters of the studied systems. First, for each RAS parameter, we study the code that checks the parameter setting at the system’s initialization phase2 (if any) and the code that later uses the parameter’s value. Then, we compare these two sets of code (checking versus usage) and examine if the initial checking is sufficient to detect configuration errors. If an error can escape from the initialization phase and break the usage code, it is a potential LC error.

We verify each LC error discovered from source code by exposing and observing the impact of the error. We first inject the errors into the system’s configuration files and launch the system; then we trigger the manifestation conditions to expose the error impact. For example, to verify the LC errors in the HDFS auto-failover feature, we start HDFS with the erroneous fail-over settings, trigger the fail-over procedure by killing the active Node, and examine if the fail-over can succeed. As all the LC errors are verified through their manifestation, there is no false positive in the reported numbers.

2.2 Findings

Finding 1: Many (14.0%–93.2%) of the studied RAS parameters do not have any special code for checking the correctness of their settings. Instead, the correctness is verified (implicitly) when the parameters’ values are actually used in operations such as a file open call.

Table 4 shows the number of the studied RAS parameters that rely on the usage code for verifying correctness, because their initial checks are either missing or incomplete. Most of the studied RAS parameters in HDFS, YARN, and HBase do not have any special code for checking the correctness of their settings. These systems

2 A system’s initialization phase is defined from its entry point to the point it starts to serve user requests or workloads.
Auto-failover configuration parameters: 
- dfs.ha.fencing.ssh.connect-timeout
- dfs.ha.fencing.ssh.private-key-files

1. LC Errors:
   - Ill-formed numbers (e.g., typos) for ssh timeout;
   - Invalid paths for private-key files (e.g., non-existence, permission errors).

2. Initial checks: None.

3. Late execution:
   - Parse the timeout setting to an integer value;
   - Read the file specified by the key-files setting.

```java
public boolean tryFence(...) {
    ...timeout = getInt("dfs.ha.fencing.ssh.connect-timeout");
    ...session.createSession();
    ...}

fis = new FileInputStream("dfs.ha.fencing.ssh.private-key-files");
```

4. Manifestation:
   - IllegalArgumentException (when parsing timeout to an integer)

5. Consequence:
   - HDFS auto-failover fails, and the entire HDFS service becomes unavailable.

(a) Missing initial checking

Error-handling configuration parameter: Apache httpd-2.4.10
CoreDumpDirectory

1. LC Errors:
   - The running program has no permission to access coredump directory.

2. Initial checks: Check if the path points to an existent directory.
   - if (finfo.filetype != APR_DIR)
   - if (finfo.isDirectory)

3. Late execution: Change working directory (chdir) to the path.

```java
apr_filepath_set(...)
if (!apr_errno)
return CoreDumpDirectory is not a directory;
```

4. Manifestation:
   - Error code returned by the chdir call

5. Consequence:
   - Apache httpd cannot switch to the configured directory, and thus fails to generate the coredump file upon server crashing.

(b) Incomplete initial checking

Figure 3: New LC errors discovered in the latest versions of the studied software, both of which are found to have caused real-world failures [40, 41]. For all these LC errors, the correctness checking is implicitly done when the parameters’ values are actually used in operations, which is unfortunately too late to prevent the failures.

Table 5: The studied configuration parameters whose values are not used at the system’s initialization phase.

<table>
<thead>
<tr>
<th>Software</th>
<th>Not used during initialization</th>
<th>Studied param.</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS</td>
<td>17 (38.6%)</td>
<td>44</td>
</tr>
<tr>
<td>YARN</td>
<td>9 (25.7%)</td>
<td>35</td>
</tr>
<tr>
<td>HBase</td>
<td>3 (12.0%)</td>
<td>25</td>
</tr>
<tr>
<td>Apache</td>
<td>4 (28.6%)</td>
<td>14</td>
</tr>
<tr>
<td>Squid</td>
<td>4 (19.0%)</td>
<td>21</td>
</tr>
<tr>
<td>MySQL</td>
<td>6 (13.9%)</td>
<td>43</td>
</tr>
</tbody>
</table>

Table 5 shows another new LC error we discovered. In this case, though the initial checking code covers file existence and types, it misses other constraints such as file permissions. This leaves Apache subject to permission-related LC errors (which is reported as one common cause of core-dump failures upon server crash [41]).

As shown by Figure 3b, one configuration parameter could have multiple subtle constraints depending on how the system uses its value. For example, a configured file path used by chdir has different constraints from files accessed by open: even for files accessed by the same open call, different flags (e.g., O_RDONLY versus O_CREAT) would result in different constraints. Implementing code to check such constraints is tedious and error-prone.

Finding 2: Many (12.0%–38.6%) of the studied RAS configuration parameters are not used at all during the system’s initialization phase.

Table 5 counts the studied configuration parameters that are not used at the system’s initialization phase, but are consumed directly in late execution (e.g., when dealing with failures). Figure 3a is such an example. Since all these parameters are from RAS features, it is natural for their usage to come late on demand.

Some Java programs put the checking or usage code of the parameters in the class constructors, so that the errors can be exposed when the class objects are created (specialy, this is used as the practice for quickly fixing LC errors [18, 19, 54]). However, this may not fundamentally avoid LC errors if the class objects are not created during the system’s initialization phase.

Note: RAS configurations can be implemented with early usage at the system’s initialization phase. As shown in Table 5, the majority of RAS configurations are indeed used during initialization. For example, all the studied systems choose to open error-log files at initialization time, rather than waiting until they have to print the error messages to the log files upon failures.

break the fail-over procedure upon the NameNode failures (as the values are not checked or used early), making the entire HDFS service become unavailable.

Apache, MySQL, and Squid all apply specific configuration checking procedures at initialization, mainly for checking data types and data ranges. However, for more complicated parameters, some checking is incomplete. Figure 3b shows another new LC error we discovered. In this case, though the initial checking code covers file existence and types, it misses other constraints such as file permissions. This leaves Apache subject to permission-related LC errors (which is reported as one common cause of core-dump failures upon server crash [41]).
Finding 3: Resulting from Findings 1 and 2, 4.7%–38.6% of the studied RAS parameters do not have any early checks and are thereby subject to LC errors which can cause severe impact on the system’s dependability.

Table 6 shows the number of the RAS configuration parameters that are subject to LC errors in the studied systems. 11 of these parameters have been confirmed/ fixed by the developers after we reported them.

### 3 PCHECK Design and Implementation

PCHECK is a tool for enabling early detection of configuration errors for a given systems program. The objective of PCHECK is to automatically generate configuration checking code (called checkers) based on the original program, and invoke them at the system initialization phase, in order to detect LC errors.

PCHECK tries to generate checkers for every configuration parameter. It is not specific to RAS configurations and has no assumption on the existence of any LC errors. The checker of a parameter emulates how the system uses the parameter’s value in the original execution, and captures anomalies exposed during the emulated execution as the evidence of configuration errors.

PCHECK is built on top of the Soot [3] and LLVM [4] compiler frameworks and works for both Java and C system programs. PCHECK works on the intermediate representations (IR) of the programs (LLVM IR or Soot Jimple). It takes the original IR as inputs, and outputs the generated checkers, and inserts them into bytecode/bytecode files (which are then built into native binaries). This may require prepending the build process by replacing the compiler front-end with Soot or Clang [47].

PCHECK faces three major challenges: (1) How to automatically emulate the execution that uses configuration values? (2) Since the checkers will be inserted into the original program and will run in the same address space, how does one make the emulation safe without incurring side effects on the system’s internal state and external environment? (3) How to capture anomalies during the emulated execution as the evidence of configuration errors (the emulation alone cannot directly report errors)?

To address the first challenge, PCHECK extracts the instructions that transform, propagate, and use the value of every configuration parameter using a static taint tracking method. PCHECK then makes a best effort to produce the context (values of dependent variables) necessary for emulating the execution. The extracted instructions, together with the context, are encapsulated in a checker.

For the second challenge, PCHECK “sandboxes” the auto-generated checkers by rewriting instructions that would cause side effects. PCHECK avoids modifications to global variables by copying their values to local ones, and rewrites the instructions that may have external side effects on the underlying OS.

To address the third challenge, PCHECK leverages system- and language-level error identifiers (including runtime exceptions, system-call error codes, and abnor-
1. Source code: MySQL 5.7.6

```c
bool flush_error_log() { ...
    redirect_std_streams(log_error_file); ...
} /*src/log.cc*/

static bool redirect_std_streams(char* file) {
    reopen_fstream(file, ...., stderr);
} /*src/log.cc*/

my_bool reopen_fstream(char* filename, ...., FILE *errstream) {
    ... my_freopen(filename, ["a", errstream); ...
} /*src/log.cc*/

FILE *my_freopen(char *path, char *mode, FILE *stream) {
    result = freopen(path, mode, stream);
} /*sys/my_fopen.c*/
```

2. Generated checker (simplified for clarity):

```c
bool check_log_error() {
    char* mode = "a";
    bool check_util_freopen(char*, mode, stream);
    bool cr = check_util_freopen(log_error_file, mode);
    if (cr == false) {
        fprintf(stderr, "log_error is misconfigured.");
    } return cr;
}
```

Figure 4: Illustration of PCHECK’s checker generation (using a real-world LC error example [26]). PCHECK replaces the original call (fopen) with check utilities based on access and stat to prevent side effects (§3.2). To execute the instructions, the necessary execution context needs to be produced. Note that we illustrate the checker using C code for clarity; the actual code is in LLVM IR or Soot Jimple.

3.1 Emulating Execution

To emulate the execution that uses a configuration parameter, PCHECK first identifies instructions that load the parameter’s value into program variables (§3.1.1). Starting from there, PCHECK performs forward static analysis to extract all the instructions whose execution uses the parameter’s value, and hence are the candidates to be included in the checkers (§3.1.2). It then analyzes backwards to figure out the values of dependent variables in these instructions, as the execution context (§3.1.3). Finally, PCHECK composes checkers using the above instructions and their context (§3.1.4).

Note that the emulation does not need to include the conditions under which the configurations are used. Instead, it focuses on executing the instructions that consume the configuration values—the goal is to check if using the configuration would cause any anomalies when its value is needed. With this design, PCHECK is able to effectively handle large, non-deterministic software programs, without the need to inject/simulate hard-to-trigger error conditions under which LC errors are exposed.

3.1.1 Identifying Starting Points

As the configuration-consuming execution always starts from loading the configuration value, PCHECK needs to identify the program variables that initially store the value of each parameter, as the starting points.

PCHECK adopts the common practices presented in previous work [8, 22, 32, 33, 52, 60, 61] to obtain the mapping from configuration parameters to the corresponding variables. The basic idea is to let developers specify the interface\(^5\) for retrieving configuration values, and then automatically identify program variables that load the values based on the interface. As pointed out by [52], most mature systems have uniform configuration interfaces for the ease of code maintenance. For instance, to work with HDFS, PCHECK only needs to know the configuration getter functions (e.g., getInt and getString in Figure 3a) declared in a single Java class; identifying them only requires several lines of specifications using regular expressions. In general, specifying interface requires little specification efforts, compared to annotating every single variable for a large number of configuration parameters. In the evaluation, the specifications needed for most systems are less than 10 lines (§4: Table 7).

\(^5\) The interface could be APIs, data structures, or parsing functions [52, 35]. It is reported that only three types of interfaces are commonly used to store/retrieve configurations [52, 35].
3.1.2 Extracting Instructions Using Configurations

For each configuration parameter, PCHECK extracts the instructions that propagate, transform, and use the parameter’s value using a static taint tracking method. For a given parameter, the initial taints are the program variables that store the parameter’s value (§3.1.1). The taints are propagated via data-flow dependencies (including assignments, type casts, and arithmetic/string operations), but not through control-flow dependencies to avoid over-tainting [39]. All the instructions containing taints are extracted, and will be encapsulated in a checker.

Note that one parameter could be used in multiple execution paths, and thus have multiple checkers. We explain how multiple checkers are aggregated in §3.1.4.

Ordinarily, the extracted instructions from data-flow analysis do not include branches. However, if a tainted instruction is used as a branch condition whose branch body encloses other tainted instructions, PCHECK performs additional control-flow analysis to retain the control dependency of these instructions. One pattern is using a configuration value p after a null-pointer check, in the form of, if (p != NULL) { use p; }. PCHECK recovers the conditional branch and ensures that if p’s value is NULL, the instructions using p inside the branch would not be reached. Moreover, PCHECK checks if a tainted branch condition leads to abnormal program states, for which it inserts error-reporting instructions (see §3.3).

The taint tracking is inter-procedural, context sensitive, and file sensitive. Inter-procedure is necessary because configuration values are commonly passed through procedure calls, as illustrated in Figure 4. We adopt a summary-based inter-procedural analysis, and assemble the execution based on arguments/returns. PCHECK maintains the call sites; thus it naturally enables context sensitivity which helps produce context by backtracking from callees to callers (c.f., §3.1.3). Field sensitivity is needed as configuration values could be stored in data structures or as class fields. PCHECK scales well for real-world software systems, as configuration-related instructions form a small part of the entire code base. We do not explicitly perform alias analysis (though it is easy to integrate), as configuration variables are seldom aliased.

3.1.3 Producing Execution Context

Some of the extracted instructions that use configuration variables may not be directly executable, if they contain variables that are not defined within the extracted instruction set. To execute such instructions, PCHECK needs to determine the values of these undefined variables (which we refer to as “dependent variables”) in order to produce self-contained context.

PCHECK will include a variable and the corresponding instructions in the emulated execution, only when this variable’s value stems from configuration values (e.g., path in Figure 4) or can be statically determined along the data-flow paths of the configuration value (e.g., mode and stream in Figure 4). PCHECK does not include dependent variables whose values come from indeterminate global variables, external inputs (from I/O or network operations such as read and recv), values defined out of the scope of the starting point, etc. For such dependent variables, PCHECK removes the instruction that uses them as operands, together with the succeeding instructions. Those variables’ values may not be available during the initialization phase of the system execution; using them would lead to unexpected results.

To produce the context, PCHECK backtracks each undefined dependent variable first intra-procedurally and then inter-procedurally (to handle the arguments of procedure calls). The backtracking starts from the instruction that uses the variable as its operand, and stops until either PCHECK successfully determines the value of the variable or gives up (the value is indeterminate). In Figure 4, PCHECK backtracks mode used by the tainted instruction and successfully obtains its value “a”.

PCHECK only attempts to produce the minimal context necessary to emulate execution for the purpose of checking. As an optimization, PCHECK is aware of how certain types of instructions will be rewritten in later transformations (e.g., for side-effect prevention, §3.2). In Figure 4’s example, PCHECK knows how the freopen call will be rewritten later. Therefore, it only produces the context of mode which is needed to check the file access; the other dependent variable stream is ignored as it is not needed for the checking.

Sometimes, the dependent variables come from other configuration parameters. PCHECK can capture the relationships among multiple configurations, e.g., one parameter’s value has to be larger or smaller than another’s.

3.1.4 Encapsulation

For each configuration parameter, PCHECK encapsulates the configuration-consuming instructions together with their context into a checker, in the form of a function. PCHECK clones the original instructions and their operands. For local variables used as operands, PCHECK clones a new local variable and replaces the original variable with the new one. If the instructions change global variables, PCHECK generates a corresponding local variable and copies the global variable’s value to the local one (to avoid changing the global program state). When it involves procedure calls, PCHECK inlines the callees.

Handling multiple execution paths. For configuration parameters whose values are used in multiple distinct execution paths, PCHECK generates multiple checkers and
aggregates their results. The configuration value is considered erroneous if one of these checkers complains.

PCHECK needs to pay attention to potential path explosion to avoid generating too many checkers. Fortunately, in our experience, configuration values are usually used in a simple and straightforward way, with only a small number of different execution paths to emulate. This makes the PCHECK approach feasible.

Moreover, PCHECK merges two checkers if they are equivalent or if one is equivalent to a subset of the other. PCHECK does this by canonicalizing and comparing the instructions in the checkers’ function bodies. Additionally, PCHECK merges checkers which start with the same transformation instruction sequence by reusing the intermediate transformation results.

Note that the checkers with no error identifiers (§3.3) or considered redundant (§3.4) will be abandoned. As shown in §4.4, the number of generated checkers are well bounded, and executing them incurs little overhead.

### 3.2 Preventing Side Effects

PCHECK ensures that the generated checkers are free of side effects—running the checkers does not change the internal program state beyond the checker function itself, or the external system environment (e.g., filesystems and OSes). Therefore, PCHECK cannot blindly execute the original instructions. For example, if the checker contains instructions that call `exec`, running the checker would destruct the current process image. Similarly, creating or deleting files is not acceptable, as the filesystem state before and after checking would be inconsistent.

**Internal side effects** are prevented by design. PCHECK ensures that each checker only has local effects. As discussed in §3.1.4, PCHECK avoids modifying global variables in the checker function; instead, it copies global variable values to local variables and uses the local ones instead. The checker does not manipulate pointers if the pointed values are indeterminate.

**External side effects** are mainly derived from certain system and library calls that interact with the external environment (e.g., filesystems and OS states). In order to preserve the checking effectiveness without incurring external side effects, PCHECK rewrites the original call instructions to redirect the calls to predefined check utilities. A check utility models a specific system or library call based on the call semantics. It validates the arguments of the call, but does not actually execute the call. PCHECK implements check utilities for standard APIs and data structures (including system calls, `libc` functions for C, and Java core packages defined in SDK). The check utilities are implemented as libraries that are either statically linked into the system’s bytecode (for C programs), or included in the system’s `classpath` (for Java programs). In Figure 4, the check utility of `freopen` checks the arguments of the call using `access` and `stat` which are free of side effects (the original `freopen` call will close the file stream specified by the third argument).

PCHECK skips instructions that `read/write` file content or `send/receiv` network packets, in order to stay away from external side effects and heavy checking overhead. Instead, PCHECK performs metadata checks for files and reachability checks for network addresses. This helps the generated checkers be safe and efficient, while still being able to catch a majority of real-world LC errors.

For any library calls that are not defined in PCHECK or do not have known side effects (e.g., some library calls would invoke external programs/commands), PCHECK defensively removes the call instructions (together with the succeeding instructions) to avoid unexpected effects.

One alternative approach to preventing external side effect is to running the checkers inside a sandbox or even a virtual machine at the system initialization phase. This may save the efforts of implementing the check utilities and rewriting system/library call instructions. However, such approach would impair the usability of PCHECK, because it requires additional setups from system administrators in order to run the PCHECK-enhanced program.

### 3.3 Capturing Anomalies

As the emulator executes the emulation that uses the configuration value, anomalies exposed during execution indicate that the value contains errors—the same problem that would occur during real execution. In this case, the checker reports errors and pinpoints the parameter.

PCHECK captures anomalies based on the following three types of **error identifiers**: (1) runtime exceptions that disrupt the emulated execution (for Java programs); (2) error code returned by system and library calls (for C programs); and (3) abnormal program termination and error logging that indicate abnormal program states.

For Java programs, PCHECK captures runtime anomalies based on Java’s `Exception` interface, the language’s uniform mechanism for capturing error events. PCHECK places the body of the checker function in a `try/catch` block. The abnormal execution would throw `Exception` objects and fall into the catch block. In this case, the checker reports errors and prints the stack traces.

C programs do not have the uniform error interfaces. Thus, PCHECK leverages the error identifiers defined by specific system/library call semantics, i.e., the return values and `errno`. For example, if the access call returns `-1`, it means the call failed when accessing the file (with the reason being encoded in `errno`). In PCHECK, we predefine the error identifiers for commonly-used system and

---

6The emulated execution paths are not the original execution paths (they only include the configuration-related instructions).
l1bc calls to decide whether a call succeeded or failed. If the call fails, the checker reports configuration errors.

In addition to the anomalies exposed by system and library APIs, a program usually contains hints of abnormal program states. Such hints are instructions such as exit, abort, throw, false assertion, error logging, etc. PCHECK treats these hints as one type of anomalies. If an instruction is post-dominated by any anomaly hints, the instruction itself indicates an abnormal state of execution. Thus, PCHECK reports configuration errors when the checker emulates such error instructions. PCHECK records these hints during the code analysis in §3.1.2, and inserts error-reporting instructions into the checker at the corresponding locations.

PCHECK abandons the checkers that do not contain any of the three types of error identifiers discussed above. In other words, running such checkers cannot expose any explicit anomalies (no evidence of configuration errors).

### 3.4 Invoking Early Checkers

Once the checkers are generated, PCHECK inserts call instructions to invoke the checkers at the program locations specified by developers. The expected location is at the end of the system initialization phase to make the checkers the last defense against LC errors.

Figure 5 shows the locations annotated for PCHECK to invoke the auto-generated checkers for Squid and HDFS. For server systems like Squid, the checkers should be invoked before the server starts to listen and wait for client requests. For distributed systems like HDFS, the checkers should be invoked before the system starts to connect and join the cluster. As all the evaluated systems fall in these two patterns, we believe that specifying the invocation locations is a simple practice for developers.

Some C programs may change user/group identities. Typically, the program starts as root and then switches to unprivileged users/groups (e.g., nobody) at the end of initialization before handling user requests. In Figure 5, the switch is performed inside mainInitialize. As the checkers are invoked in the end of the initialization, the checking results are not affected by user/group switches.

To capture the TOCTTOU errors, PCHECK also supports running the generated checkers periodically in a separate thread. Periodical checking is particularly useful for catching configuration errors that occur after the initial checking (e.g., due to environment changes such as remote host failures and inadvertent file deletion).

#### Avoiding redundant checking

PCHECK abandons the redundant checkers which are constructed from instructions that would be executed before reaching the invocation location—any configuration errors reported by such checkers should have already been detected by the system’s built-in checks, or have been exposed when the configuration value is used, before the checker is called.

**Creating standalone checking programs.** Another option to invoking the early checkers is to create a standalone checking program comprised of the checkers, and run it when the configuration file changes. This approach eliminates the need to deal with internal side effect; on the other hand, the checking program is still prohibited to have external side effect. Note that the generated checkers start from the instructions that load configuration values (§3.1.1); therefore, the checking program needs to include the procedures that parse configuration files and store configuration values. This is straightforward for the software systems with modularized parsing procedures, but could be difficult if the parsing procedures cannot be easily decoupled from the initialization phase (the initialization may have external side effects).

### 4 Experimental Evaluation

#### 4.1 Methodology

We first evaluate the effectiveness of PCHECK using the 37 new LC errors discovered in our study. As discussed in §2, all these new LC errors are from the latest versions of the systems; any of them can impair the corresponding RAS features such as fail-over and error handling.

As the design of PCHECK is inspired by the above LC errors, our evaluation contains two more sets of benchmarks to evaluate how PCHECK works beyond these errors. First, we evaluate PCHECK on a distinct set of 21 real-world LC errors that caused system failures in the past. These LC errors are collected from the datasets in prior studies related to configurations [6, 11, 51, 55, 59]; all of them were introduced by real users and caused real-world failures. Some of these cases have different code

---

Figures and code snippets have been omitted for brevity.
patterns from the ones we discovered in §2. Table 7 lists the number of LC errors in each system.

Furthermore, we apply PCHECK to 830 configuration files of the studied systems (except Squid) collected from the official mailing lists of these systems and online technical forums such as ServerFault and StackOverflow [1]. This simulates the experience of using PCHECK on real-world configuration files (§4.2). Moreover, it helps measure the false positive rate of the checking results (§4.6).

Note that we evaluate PCHECK upon all types of LC errors, instead of any specific error types. Therefore, the evaluation results indicate the checking effectiveness of PCHECK in terms of all possible LC errors. Table 8 categorizes and exemplifies the LC errors used in the evaluation based on their types.

Also, the evaluation does not use synthetic errors generated by mutation or fuzzing tools (e.g., ConfErr [23]). Most of the synthetic errors are not LC errors—they are manifested or detected by the system’s built-in checks at the system’s initialization time. Thus, using such errors would make the results less meaningful to LC errors.

For each system, we apply PCHECK to generate the early checkers and insert them in the system’s program. Table 7 lists the setup efforts for the each system evaluated, measured by the lines of specifications for identifying the start points (c.f., §3.1.1) and annotations of the invocation locations (c.f., §3.4). Then, we apply the auto-generated checkers to the configuration files that contain these LC errors. We evaluate the effectiveness of PCHECK based on how many of the real-world LC errors can be reported by the auto-generated checkers.

We compare the checking results of PCHECK with conf_spellchecker [2, 35], a state-of-the-art static configuration checking tool built on top of automatic type inference of configuration values [33, 35]. For each defined type, conf_spellchecker implements corresponding checking functions which are invoked to check the validity of the configuration settings.

### 4.2 Detecting Real-world LC Errors

PCHECK detects 70+% of both historical and new LC errors (as shown in Table 9), preventing the latent manifestation and resultant system damage imposed by these errors. The results are promising, especially considering that we evaluate PCHECK using all types of configuration errors instead of any specific type. Indeed, PCHECK is by design generic to any types of configuration errors that can be exposed through execution emulation. Many of these LC errors cannot be detected by the state-of-the-art detection tools, as discussed below and in §4.3.

Among the different types of LC errors, PCHECK detects all the errors violating the types/formats and options/ranges constraints. These two types of errors usually go through straightforward code patterns and do not have dependencies with the system’s runtime states. For example, most type/format errors in HDFS and YARN are manifested when these systems read and parse the configuration files. As the auto-generated checkers invoke the getter functions, it triggers exceptions and detects the errors.

PCHECK detects the majority of LC errors that violate file-related constraints (including special files such as directories and executables). We observe that the majority of the file parameters fall into recognized APIs, such as open, fopen, and FileInputStream. The undetected file-related LC errors are mainly caused by (1) unknown external usage and (2) indeterminate context. The former prevents the generated checkers from being executed, and the latter stops generation of the checkers. For example, some errors reside in parameters whose values are concatenated into shell command strings, used as the argument of system() (to invoke /bin/sh to execute the command). As PCHECK has no knowledge of any
shell commands, it removes the `system()` call because the side effects are unknown. The other undetected errors are in directories or file prefixes which are merged with dynamic contents from user requests which cannot be obtained statically; thereby, the corresponding checkers cannot be generated. These two causes (unknown external usage and indeterminate context) also account for the undetected errors in the “other” category.

In general, PCHECK is effective in checking errors that are manifested through execution anomalies with error identifiers defined in §3.3, such as those failing at system/library calls or throwing exceptions in the controlled branch. Whereas, it is hard for PCHECK to detect errors defined by application-specific semantics, such as email addresses, internal error code, etc.

We apply `conf_spellchecker` on the same sets of LC errors. Compared with PCHECK, `conf_spellchecker` detects 7 (33.3%) and 11 (29.7%) less LC errors in the historical and new error benchmarks, respectively. The main reason for PCHECK’s outperformance is that the execution emulation can achieve fine-grained checking towards high fidelity to the original execution. For example, `conf_spellchecker` can only infer the type of a configuration setting to be a “File”. However, it does not understand how the system accesses the file in the execution. Thus, it reports errors if and only if “the file is neither readable nor writable” [2]. This heuristic would miss LC errors such as read-only files to be written by the system. Furthermore, type alone only describes a subset of constraints. `conf_spellchecker` misses the LC errors that violate other types of constraints such as data ranges.

### 4.3 Checking Real-world Configuration Files

We apply the checkers generated by PCHECK to 830 real-world configuration files. PCHECK reports 282 true configuration errors and three false alarms (discussed in §4.6). As shown in Table 10, many (37.5%–87.8%) of the reported configuration errors can only be detected by considering the system’s native execution environment. These configuration settings are valid in terms of format and syntax (in fact, they are likely to be correct in the original hosts). However, they are erroneous when used on the current system because the values violate environment constraints such as undefined environment variables, non-existent file paths, unreachable IP addresses, etc. Since PCHECK emulates the execution that uses the configuration values on the system’s native execution environment, it naturally detects these errors. On the other hand, such configuration errors are not likely to be detected by traditional detection methods [29,31,36,46,57] that treat configuration values as string literals, and thus are agnostic to the execution environment.

<table>
<thead>
<tr>
<th>Software</th>
<th># config files</th>
<th># (%) detected config. errors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>All</td>
</tr>
<tr>
<td>HDFS</td>
<td>245</td>
<td>40</td>
</tr>
<tr>
<td>YARN</td>
<td>81</td>
<td>49</td>
</tr>
<tr>
<td>HBase</td>
<td>405</td>
<td>139</td>
</tr>
<tr>
<td>Apache</td>
<td>65</td>
<td>41</td>
</tr>
<tr>
<td>MySQL</td>
<td>34</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 10: Configuration errors detected by applying the checkers on real-world configuration files. Many of the errors can only be detected by considering the system’s native environment (§4.3).

### 4.4 Checker Generation

Table 11 shows the number of configuration parameters that have checkers generated by PCHECK and the total number of generated checkers for the evaluated systems (multiple checkers could be generated for a parameter).

<table>
<thead>
<tr>
<th>Software</th>
<th># checked param. (# checkers)</th>
<th>All params</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS</td>
<td>164 (232)</td>
<td>164</td>
</tr>
<tr>
<td>YARN</td>
<td>116 (200)</td>
<td>116</td>
</tr>
<tr>
<td>HBase</td>
<td>125 (201)</td>
<td>125</td>
</tr>
<tr>
<td>Apache</td>
<td>18 (41)</td>
<td>97</td>
</tr>
<tr>
<td>Squid</td>
<td>45 (74)</td>
<td>216</td>
</tr>
<tr>
<td>MySQL</td>
<td>32 (51)</td>
<td>462</td>
</tr>
</tbody>
</table>

Table 11: The number of parameters with checkers generated by PCHECK and the total number of generated checkers (each represents a distinct parameter usage scenario).
4.5 Checking Overhead

The checkers are only invoked at the initialization phase or run in a separate thread, thus they have little impact on the systems’ runtime performance. We measure their overhead to be the time needed to execute these checkers, by inserting time counters before and after invoking all the checkers. Table 12 shows the time in milliseconds (ms) to run the checkers on a 4-core, 2.5GHz processor connected to a local network (for distributed systems like HDFS, YARN, and HBase, the peer nodes are located in the same local network). The checking overhead for Apache and MySQL is negligible (less than 5ms); Squid needs around 100ms because it has a parameter that points to public IP addresses (announce_host). The overhead for the three Java programs is less than a second. The main portion of the time is spent on network- and file-related checking. Since PCHECK only performs lightweight checks (e.g., metadata checks and reachability checks), the overhead is small. Note that the checkers are currently executed sequentially. It is straightforward to invoke multiple checkers in parallel to reduce overhead, as all the checkers are independent.

4.6 False Positives

We measure false positives by applying the checkers generated by PCHECK to both the default configuration values of the evaluated systems and the 830 real-world configuration files, and examine whether or not our checkers would falsely report errors. We also manually inspect the code of the generated checkers in LLVM IR and Jimple to look for potential incorrectness.

Among all the configuration parameters in the evaluated systems, only three of them have false alarms reported by the auto-generated checkers: two from YARN and one from HBase. All these false positives are caused by the checkers incorrectly skipping conditional instructions affected by the configuration value (§3.1.2), due to unsound static analysis that misses control dependencies. This results in emulating the execution that should never happen in reality—certainly, the anomalies exposed in such execution are unreal. The overall false positive rates are low. YARN has the most configuration parameters with false checkers, with the false positive rate of 1.7% (2 over 116 parameters). Note that checkers with false positives can be removed by the developers or disabled by the administrators in the field (c.f., §3: Usage).

5 Limitations

No tool is perfect. PCHECK is no exception. Like many other error detection tools, PCHECK is neither sound nor complete for its checking scope and the design trade-offs.

PCHECK targets on the specific type of configuration errors which are manifested through explicit, recognizable instruction-level anomalies (c.f., §3.3). It cannot detect legal misconfigurations [55] that have valid values but do not deliver the intended system behavior. The common legal misconfigurations include inappropriate configuration settings that violate resource constraints or performance requirements (e.g., insufficient heap size and too small timeout). Such misconfigurations are notoriously hard to detect and are often manifested in a latent fashion as well, such as runtime out-of-memory errors [10] (resources are not used up immediately). However, detecting resource- and performance-related misconfigurations would need dynamic information regarding resource usage and performance profiling, which is beyond the static methods of PCHECK.

In addition, PCHECK cannot emulate the execution that depends on runtime inputs/workloads, or does not have statically determinate context in the program code (c.f., §3.1.3). Thus, it would miss the configuration errors that are only manifested during such execution. Nevertheless, indeterminate context (e.g., those derived from inputs and workloads) can potentially be modeled with representative values, which could significantly improve the capability of checker generation.

One design choice we make is to trade soundness for safety and efficiency—PCHECK aims to detect common LC errors without incurring side effects or much overhead. For example, PCHECK does not look into file contents but only checks if the file can be accessed as expected. Similarly, PCHECK only checks the reachability of a configured IP address or host instead of connecting and sending packets to the remote host. It is possible that certain sophisticated errors can escape from PCHECK (e.g., the configured file is corrupted and thus has wrong contents). As the first step, we target on basic, common errors, as they already account for a large number of real-world LC errors [24, 43, 55]. Efficiently detecting sophisticated errors may require not only deeper analysis but also application semantics.

6 Concluding Remarks

This paper advocates early detection of configuration errors to minimize failure damage, especially in cloud and data-center systems. Despite all the efforts of validation,
review, and testing, configuration errors (even those obvious errors) still cause many high-impact incidents of today’s Internet and cloud systems. We believe that this is partly due to the lack of automatic solutions for cloud and data-center systems to detect and defend against configuration errors (the existing solutions are hard to be applied, due to their strong reliance on datasets).

We envisage that PCHECK is the first step towards a generic and systematic solution to detect configuration errors. PCHECK does not require collecting any external datasets and is not specific to any specific rules. It detects configuration errors based on how the system actually uses the configuration values. With PCHECK, we demonstrate that such detection method can effectively detect the majority (75+%) of real-world LC errors, with little runtime overhead and setup effort.

7 Acknowledgement

We greatly appreciate the anonymous reviewers and our shepherd, Peter M. Chen, for their insightful comments and feedback. We thank the Opera group, the UCSD Systems and Networking group, and Shelby Thomas for useful discussions and paper proofreading. Tao Cai participated in the implementation of PCHECK. Lijiong Yang contributed to the study of RAS related configuration parameters. Yuanyuan Zhou’s group is supported in part by NSF grants (CCR-1526966, CCR-1321006), and a gift from Facebook, and supports from NetApp. Shan Lu’s research is supported in part by NSF grants (IIS-1546543, CNS-1563956, CNS-1514256, CCF-1514189, CCF-1439091), and generous supports from Alfred P. Sloan Foundation and Google Faculty Research Award.

References


[38] Squid Bug #4186. The mail notification feature is buggy and does not deal with configuration errors. http://bugs.squid-cache.org/show_bug.cgi?id=4186.


[54] YARN ISSUE #2166. Timelineserver should validate that yarn.timeline-service.leveldb-timeline-store.ttl-interval-ms is greater than zero when level db is for timeline store. https://issues.apache.org/jira/browse/YARN-2166.


