Multi-misconfiguration Diagnosis via Identifying Correlated Configuration Parameters

Yingnan Zhou, Xue Hu, Sihan Xu*, Yan Jia*, Yuhao Liu, Junyong Wang, Guangquan Xu, *Member, IEEE*, Wei Wang, Shaoying Liu, *Fellow, IEEE*, Thar Baker, *Senior Member, IEEE*

Abstract—Software configuration requires that the user sets appropriate values to specified variables, known as configuration parameters, which potentially affect the behaviors of software system. It is an essential means for software reliability, but how to ensure correct configurations remains a great challenge, especially when a large number of parameter settings are involved. Existing studies on misconfiguration diagnosis treat all configurations independently, ignoring the constraints and correlations among different configurations. In this paper, we reveal the phenomenon of multi-misconfigurations and present a tool, MMD, for multi-misconfigurations diagnosis. Specifically, MMD consists of two modules: Correlated Configurations Analysis and Primary Misconfigurations Diagnosis. The former determines the correlation among each pair of configurations by analyzing the control and data flows related to each configuration. The latter is responsible for collecting a list of configurations ranked according to their suspiciousness. Combining the outputs of two modules, MMD is able to assist the user in multi-misconfigurations diagnosis. We evaluate MMD on seven popular Java projects: Randoop, Soot, Synoptic, Hdfs, Hbase, Yarn, and Zookeeper. MMD identifies 510 configuration correlations with a 4.9% false positive rate. Furthermore, it effectively diagnoses 22 multi-misconfigurations collected from StackOverflow, outperforming two state-of-the-art baselines.

Index Terms—Configuration, correlation, multi-misconfiguration, parameters, diagnosis

1 Introduction

In modern software engineering, there is a growing trend to use a large number of configuration parameters to improve software flexibility and customizability. Users are supposed to customize values of configuration parameters, which is usually in a configuration file, to fit their own needs without recompiling programs [1]. Typically, a configuration parameter is a key-value pair, where the key is a string that defines configurable software behaviors, and the value is a user-definable variable that designates the expected behaviors. As the software scale grows, so do the number of configuration parameters. For instance, Apache Hadoop [2] has more than 2200 configuration parameters.

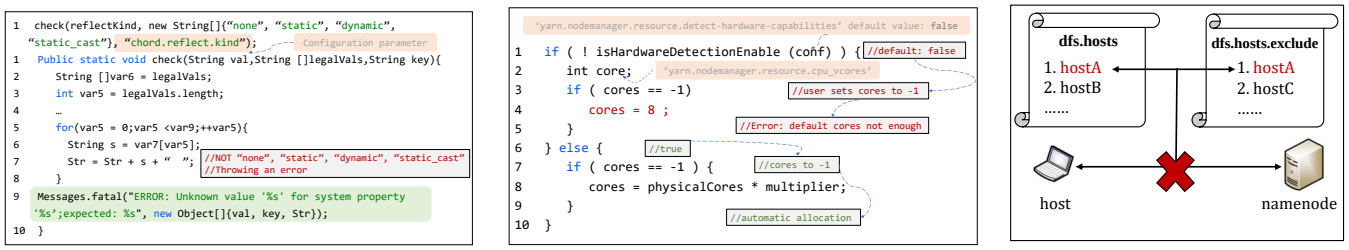
Although configuration parameters can improve the customizability of software, how to ensure the correctness of configuration remains a great challenge, especially when a large number of configuration parameters are involved. Software misconfiguration has been one of the major threats to software reliability, especially in large-scale systems [3], [4], [5]. Generally, software misconfigurations happen when configuration parameters are set by inappropriate values, leading to unexpected software behaviors and even failures. Figure 1a shows an instance in Jchord where the configuration parameter `chord.reflect.kind`, which specifies the algorithm to resolve reflection, only accepts `none`, `dynamic`, `static`, and `static_cast` as the configuration value [6]. When a user sets a value beyond the acceptable list, a fatal error is raised in JChord.

The challenges of misconfiguration diagnosis arise not only from a large number of configuration parameters but also from their complex correlations. Figure 1b illustrates a real-world misconfiguration in Yarn [7], which was caused by two configuration parameters related to resource allocation. When the configuration parameter `yarn.nodemanager.resource.cpu_vcores` is set to `-1` and `yarn.nodemanager.resource.detecthardware-capabilities` is set to `True` [7], Yarn [7] is capable of automatically allocating the number of CPU Vcores. However, if a user only sets the first configuration to `-1` and ignores the second one, i.e., using the default value `False`, Yarn can not allocate the number of CPU Vcores, a software misconfiguration occurs.

In addition to the aforementioned direct correlations, which is explicitly written in source code, there may also exist some indirect configuration correlations. Figure 1c shows an example in Hadoop Hdfs [8], where the configuration parameter `dfs.hosts` sets the path name of the file that specifies a list of hosts permitted to connect to the namenode, and `dfs.hosts.exclude` specifies the

- *Yingnan Zhou and Xue Hu are co-first authors of the article.*
- *Sihan Xu and Yan Jia are the corresponding authors.*
- *Yingnan Zhou, Yuhao Liu, Junyong Wang and Wei Wang are with the School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China.*
  E-mail: yingnanzhou@bjtu.edu.cn, 20120474@bjtu.edu.cn, junyong.wang@bjtu.edu.cn, wangwei1@bjtu.edu.cn.
- *Xue Hu and Guangquan Xu are with the Tianjin Key Laboratory of Advanced Networking (TANK), College of Intelligence and Computing, Tianjin University, Tianjin, China.*
  E-mail: huxue00@tju.edu.cn, losin@tju.edu.cn.
- *Sihan Xu and Yan Jia are with DISSec, College of Cyber Science, Nankai University, Tianjin, China.*
  Email: xusihan@nankai.edu.cn, jiay@nankai.edu.cn.
- *Shaoying Liu is with the Graduate School of Advanced Science and Engineering, Hiroshima University, Higashihiroshima 739-8511, Japan.*
  Email:sliu@hiroshima-u.ac.jp.
- *Thar Baker is with the School of Architecture, Technology and Engineering, University of Brighton, Brighton BN2 4GJ, UK E-mail: t.shamsa@brighton.ac.uk*

Manuscript received December 16, 2022; revised June 25, 2023.



(a) Misconfiguration Caused by a Single Configuration parameter

(b) Misconfiguration Caused by Multiple Configurations with Direct correlation

(c) Misconfiguration Caused by Multiple Configurations with Indirect correlation

Fig. 1: Three Types of Misconfigurations

file of the forbidden hosts. Theoretically, these two lists of hosts are supposed to be mutually exclusive. However, if a user defines the same host in both files, the host cannot be connected to the namenode, which leads to an indirect correlation and results in undesired software behaviors.

Despite the correlations among multiple configuration parameters, most studies on misconfiguration diagnosis assume that there is only one configuration parameter that leads to a failure [9], [10], [11], [12], [13]. To diagnose multi-misconfigurations, it is desirable to figure out correlated configurations to assist diagnosis. Correlations have also proven to be very common [14]. Nevertheless, the correlations of configuration parameters cannot be easily identified. **First**, not all configuration correlations are listed in user manuals. For example, the user manual of Yarn does not remind users of the correlations between different configurations. **Second**, when a failure occurs due to multi-misconfiguration, users can hardly associate the error with correlations among different configuration parameters. Although Chen et al. [15] proposed cDep to identify the dependencies (correlations) among different configuration parameters, cDep relies on the manually-defined code patterns to identify correlated configurations, which lead to high false negatives (e.g. Figure 1b and 1c) due to the limitation of expertise experience.

To address these challenge, in this paper, we propose a novel method to identify configuration correlations for multi-misconfiguration dagnosis. Specifically, we first analyze the data and control flows related to each configuration. Through program slicing we generate the profile vectors for each configuration and then calculate the distance between the vectors. By this means, we measure the correlations between multiple configurations for multi-misconfiguration diagnosis. We implement a tool named MMD (i.e., Multi-Misconfigurations Diagnosis).

MMD consists of two modules, i.e., Correlated Configurations Analysis and Primary Misconfiguration Diagnosis. During diagnosis, MMD first obtains a list of configuration parameters ranked by their suspiciousness of causing the configuration error. Then, it analyzes the data and control flows related to each configuration parameter and calculates the correlation between each pair of configurations. Finally, by reordering the outputs of two modules, MMD is able to diagnose multi-misconfigurations, taking into consideration both the suspiciousness and the correlations of configurations. MMD aims to assist software users in the diagnosis of multi-misconfigurations. Developers can integrate MMD with the software and users would get valuable feedback if multi-misconfigurations occur.

Finally, we evaluated MMD on seven real-world projects, i.e., Randoop [16], Soot [17], Synoptic [18], Hdfs [2], Hbase [2], Yarn [2], and Zookeeper [2], compared with two baselines (i.e., cDep [15] and ConfDiagnoser [10]). The experimental results show that MMD is capable of identifying 510 configuration correlations and effectively diagnosing 22 real-world multi-misconfigurations collected from StackOverflow [19], outperforming the baseline by a large margin.

In summary, this paper makes the next contributions.

- Our research has identified a previously unknown phenomenon called multi-misconfigurations, which result from configuration correlations and can potentially affect the precision of misconfiguration diagnosis. In Section 2, we provide a detailed explanation of the different categories of multi-misconfigurations that we have identified.
- In this study, we have introduced a novel approach for the methodical diagnosis of multi-misconfiguration software errors via the development of a tool, namely *Multi-Misconfiguration Diagnosis (MMD)*. The principal functionality of MMD entails performing slicing analysis of each configuration parameter according to the data and control flows, and subsequently measuring their distances to establish their correlations. When a software error occurs due to multiple misconfigurations, MMD outputs the problematic correlated configuration parameters to assist users diagnosing the error. The description of this part mentioned in Section 3
- We evaluated MMD on seven popular Java projects (i.e., Randoop, Soot, Synoptic, Hdfs, Hbase, Yarn, and Zookeeper) and a set of real-world multi-misconfigurations collected from StackOverflow [19]. The experimental results, shown in Section 4, exhibit the capability of MMD to assist users in diagnosing multi-misconfigurations.

In Section 2 we present several definitions and classifications of configuration errors. Section 3 details our approach, and Section 4.1 reveals the information of implementation. In Section 4, we evaluate the MMD and compare it with other work, cDep and ConfDiagnoser. Sections 4.1, 4.2, and

4.3 reveal the experiments setup, dataset, and evaluation, respectively. Section 5 provides a summary of the shortcomings of the proposed MMD.. Section 6 discusses related work, and Section 7 states our conclusions.

2 Background

In this section, we first define configuration correlations and multi-misconfigurations and then describe the motivation of the proposed method to diagnose multi-misconfigurations. Some topics are well-explored in software product lines (SPLs), we will declare the similarities and differences in Section 6.

2.1 Configuration Correlation

Chen et al. [15] classifies configuration dependencies (or correlation) into two types, i.e., functional dependency and behavioral dependency. The former denotes the correlations where a configuration can influence another configuration, and the latter describes the cases where a set of configurations altogether influence the same system behavior. Although some correlations could be successfully identified, cDep relies on manually defined code patterns and ignores correlations that are not directly reflected in the source code. Next, we classify configuration correlations into two groups and present the method to automatically estimate correlations based on the classification.

2.1.1 Direct configuration correlations

We define direct configuration correlations as those that can be directly captured from source code, and further classify these correlations into three groups, i.e., control flow correlation, data flow correlation, and functional correlation.

Control flow correlation. If the value of one configuration determines whether the other one will be executed or not, a control flow correlation occurs between two configuration groups. Figure 2 shows an example of the control flow correlation for two configurations, where `this.minDiskCheckGapMs` and `this.diskCheckTimeout` are two variables that store the values of `dfs.datanode.disk.check.min.gap` and `dfs.datanode.disk.check.timeout`, respectively. It can be seen that `this.minDiskCheckGapMs` determines whether `this.diskCheckTimeout` will be executed, which relationship could be captured in control flows. In many cases, a single configuration parameter is able to control the values of multiple configurations, forming a one-to-many control flow correlation. Just as some "enable" configuration parameters will determine whether a series of configurations are enabled or not.

Data flow correlation. If the value of one configuration parameter is influenced by the value of the other one, a data flow correlation occurs between them. Figure 3 shows an example of the data flow correlation, where `this.heartbeatInterval` and `this.stateInterval` are two variables that store the values of the configuration parameters `dfs.heartbeat.interval` and `dfs.namenode.stale.datanode.interval`, respectively. if the value of `this.heartbeatInterval` is changed, `this.stateInterval` will change too, which relationship could be captured in data flows.

```
1 if ( this.minDiskCheckGapMs <  0L ) {
2     …  // DiskErrorException for Invalid value
3  } else {          'dfs.datanode.disk.check.min.gap'
4     if  ( this.diskCheckTimeout  <  0L ) {
5        … // DiskErrorException for Invalid value
6     } else {        'dfs.datanode.disk.check.timeout'
7        … // Complete disk check
8    }                          //Boolean expression for gap&timeout
9  }                            control flow correlation
```

Fig. 2: An Example of the Control Flow Correlation

```
1 this.Expire =(2*this.heartbeatRecheck)+10000L*this.heartbeatInterval;
2 this.staleInterval = getStaleIntervalFromConf(conf, this.Expire);
'dfs.namenode.stale.datanode.interval'        'dfs.heartbeat.interval'
  //Numerical change for heartbeat&datanode
  data flow correlation
```

Fig. 3: An Example of the Data Flow Correlation

Functional correlation. Besides the correlation where one configuration influences the other one, in some cases, multiple configurations work together to provide software functionality. Figure 4 shows an example of functional correlation, where `this.heartbeatInterval` and `this.namenodeReplication` are two variables that store the values of the configuration parameters `dfs.heartbeat.interval` and `dfs.namenode.replication.interval`, respectively. It can be seen that although these configurations are not influenced by each other, the combination of their values determines the value of the variable `sleeptime`, which influences the function of moving blocks. As illustrated in this example, functional correlation is usually captured on both data and control flows.

```
1 long sleeptime =this.heartbeatInterval*2000L+this.namenodeReplication*1000L;
2 connectors = newNameNodeConnectors(…,sleepltime,…,…);
3 While (connectors.size() > 0) {          'dfs.namenode.replication.interval'
                  'dfs.heartbeat.interval'
4     // move blocks
5     Thread.sleep(sleeptime);
6  }    //heartbeat & replication → "sleeptime" Control the move block function
```

Fig. 4: An Example of the Functional Correlation

2.1.2 Indirect configuration correlations

In addition to the configuration correlations that can be directly observed from source code, there are also indirect ones hidden in the logic of software behaviors. Figure 1c is an example of such a case. Although this kind of correlation is more complex or undefined in source code, the logical functions of software also are reflected through data and control flows.

In other words, all the above correlations are reflected in data and control flows. Therefore, we could abstract configuration parameters in data and control flows to estimate the correlations.

2.2 Multi-misconfigurations

Previous studies treat each configuration parameter independently and consider software misconfigurations caused only by one configuration [10], [12], [13]. However, due to the correlations described in Section 2.1, some misconfigurations cannot be fixed only by one repaired configuration. Figure 1b shows an example of multi-misconfiguration. Based on the comparison of execution paths, ConfDiagnoser [10] is able to assign the configuration parameter `yarn.nodemanager.resource.cpu_vcores` a high score of suspiciousness. However, due to the violation of the correlation between `yarn.nodemanager.resource.cpu_vcores` and `yarn.nodemanager.resource.detect-hardwarecapabilities`, the multi-misconfiguration can not be fixed only by `yarn.nodemanager.resource.cpu_vcores` be corrected. Users need to locate both configurations and fix them to satisfy their correlations. Since the second configuration has no effect on the execution path, ConfDiagnoser fails to identify it and the misconfiguration remains unresolved.

As a consequence, identifying correlations among multiple configurations is a fundamental step into diagnose the aforementioned multi-misconfigurations. To the best of our knowledge, only Chen et al. [15] proposed cDep to identify dependencies in configurations. They manually defined a set of code patterns and utilized static taint analysis to identify dependencies. However, cDep relies on manually-defined code patterns to identify dependencies, which is limited by expert expertise. On the other hand, cDep only takes direct correlations into consideration, ignoring indirect correlations, as in Figure 1c. Finally, cDep may also ignore some direct correlations reflected in source code (e.g., the case in Figure 1b) due to incomplete definition of the code pattern and prevent over-contamination.

Motivation example. To overcome the aforementioned limitations and identify the common functionalities of different configurations, we propose a novel method that combines the statistic method and program slicing to identify configuration correlations. Figure 5 shows a part of inter-procedural control flow graph related to the configuration parameters mentioned in Figure 1b and Figure 1c. First, it can be seen that while the configuration `dfs.host` and `dfs.host.exclude` have no direct correlations such as the control flow or data flow dependencies, they share many functions (e.g., `registerDatanode`, `getHostName`, and `refreshNodes`) in their execution paths. The reason is that `dfs.host` and `dfs.host.exclude` operate similar functionalities that manage the hosts to connect to the namenode. It can also be seen that the configuration `heartbeatExpireInterval` has only one common function with `dfs.host.exclude`, indicating that there are few overlapping functionalities between them. For this reason, we propose a statistic-based method of multi-misconfiguration diagnosis, which exploits program slicing to extract the control flow and data flow information for a configuration parameter and then mines the configuration correlations based on their similarities. Given a list of suspicious configurations, the proposed method reorders them according to their suspiciousness scores and the correlations among multiple configurations. A higher level of suspicion indicates a greater likelihood of causing a configuration error, while a lower level of suspicion suggests a lower likelihood.

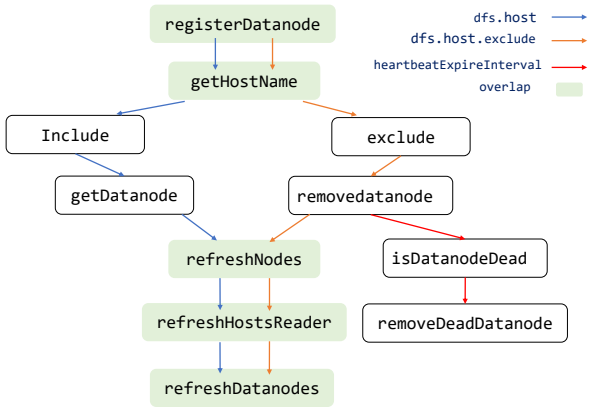


Fig. 5: An illustration of the correlations between two configurations `dfs.hosts` and `dfs.hosts.exclude`

3 Approach

In this section, we present MMD, a statistically-based tool for the automatic diagnosis of multi-misconfigurations. Its workflow is first illustrated, followed by a description of the two main modules, i.e. Correlated Configurations Analysis and Primary Misconfigurations Diagnosis.

3.1 System Overview

Figure 6 illustrates the workflow of MMD, which consists of two main modules: *Correlated Configurations Analysis (CCA)* and *Primary Misconfigurations Diagnosis (PMD)*. The CCA module takes the source code of the target software as input to analyze the correlations of configuration parameters and generates a list of correlated configurations. When a configuration error occurs, the PMD module calculates the suspiciousness of all configuration parameters and outputs a ranked list of configurations in descending order of their suspiciousness. Combining the outputs of both modules (i.e., CCA and PMD), MMD reorders the configuration parameters and assists users in multi-misconfigurations diagnosis.

Specially, during preprocessing, MMD first identifies the configuration class and locates the reading points of configuration parameters. Then, using the reading points as the seeds, the CCA module performs program slicing, extracts the block structures, and generates a vector for each configuration. Next, it calculates the distance between the representations of configurations to obtain a list of correlated configurations. In the phase of misconfiguration diagnosis, when a configuration error occurs, the PMD module calculates the suspiciousness of each statement affected by configuration, slicing backward in descending order to find configuration parameters and obtains a list of them ranked in terms of suspiciousness. Finally, MMD combines the outputs of PMD and CCA, and reorders the checklist of

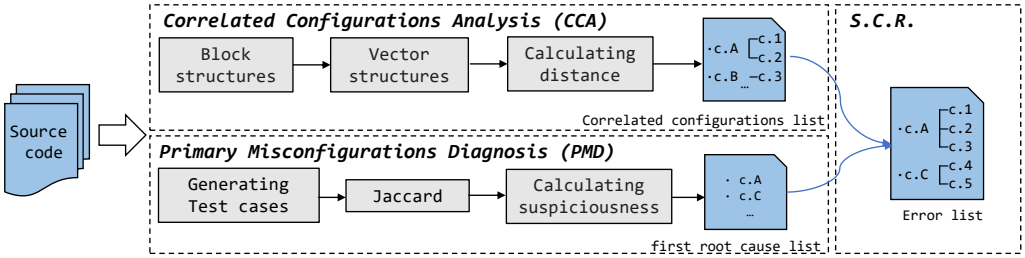


Fig. 6: the Workflow of MMD.
The abbreviation **c.** refers to a configuration parameter
The abbreviation **S.C.R.** refers to the step of Suspicious Configurations Recommendation

configuration parameters, according to the suspiciousness and correlations of configurations as an error list. In the case that the error still exists, MMD recommends a correlated configuration to the user who fixes a configuration parameter with high suspiciousness, which may be a second root cause of the misconfiguration.

3.2 Correlated Configurations Analysis

Considering the correlations among different configurations, MMD identifies the correlated configurations in the CCA module. As discussed in Section 2, correlated configuration parameters with the same functionality usually share more execution paths than unrelevant configurations. Based on this intuition, we extract the code statements influenced by each configuration and generate a vector for each configuration. By calculating the distance between a pair of configurations, we estimate the degree of the correlation between configuration parameters. The CCA module generates a static library through a one-time correlation analysis, and the resulting library can be reused as long as the code remains unchanged.

Getting the reading points of configuration parameters as the seeds, we perform program slicing and obtain a set of code statements to generate a vector for each configuration. However, there are often too many statements influenced by a configuration, resulting in very high-dimensional vectors. For instance, in Hdfs there are nearly 30,000 sliced statements for the configuration `dfs.heartbeat.interval` [8], which brings a great challenge to effectively estimate the distances among configurations.

3.2.1 Block structure extraction

To reduce the dimensions of configuration vectors while maintaining effectiveness, we optimize the notion of basic block [20], which indicates a straight-line code sequence with no branches. Specifically, we merge these methods into a single block, which are not directly relevant to the implementation of the functionality and do not differentiate the execution path of the configuration parameters. These methods are invoked by all configuration parameters, such as variable assignment (e.g., getInt, etc.). And these methods are not to call other methods within the software, except for library functions like java methods. Additionally, they are not directly related to the core functions of the software, such as logging and exception handling. These methods have a relatively minimal impact on the execution path of the functionality, as most of the configuration parameters invoke them without significant distinction in the distance calculation.

```
public int[] getInts(String name) {
  int[] ints = new int[name.length()];          block1
  for(int i = 0; i < strings.length; ++i) {
    ints[i] = Integer.parseInt(strings[i]);     block2
  }
  return ints;                                  block3
}                                               Single Block
```

Fig. 7: Example of a Get Method Construct into a Single Block

Figure 7 shows a method `getInts`, which was written to get a value of the type `int` in Hadoop. In the first basic block of `getInts`, it sequentially declares a variable. The second block is a `loop` structure, which assigns values to these variables. The third block declares the `return` statement. It can be seen that the method `getInts` is a method that performs a simple functionality for variable assignment. It only invokes Java methods and is called by all configuration parameters. While there may be variations in assignment for different types, the function primarily focuses on assignment. For this reason, we merge three basic blocks into a single block to address the high-dimensional problem of configuration vectors. For instance, given the running example in Figure 1b with cpu_vcores and hardwarecapabilities, the CCA module takes the source code of Yarn as input and extracts the block structure to represent the source code.

3.2.2 Vector representation

After obtaining the aforementioned block structures for the target software, we then represent the configurations with the control and data flows influenced by them. Procedure dependency graph is represented as $PDG = (V, D, C, S, E)$, where $V = \{n_i | i \in 0, ..., n\}$ is node (can be basic block), $D = \{n_i \rightarrow n_j | i, j \in 0, ..., n\}$, $\rightarrow$ denotes the direction of the data dependency edge, $C = \{n_i \rightarrow n_j | i, j \in 0, ..., n\}$, $\rightarrow$ denotes the direction of the control dependency edge, S is the entry of program and E is the exit of program. The PDGs are connected at call sites, consisting of a call node cn that is connected with the entry

node e of the called procedure through a call edge cn→ cn.e. All of these form system dependency graph, which can be used for slicing.

Specifically, using target software and the read point of a configuration parameter, we conduct program slicing and obtain n blocks (b_i). We use $block_i$ to denote the number of statements in the block ith. Slicing is represented as $Slice = (seed, cop, dop, v)$, where $seed \in V$, $cop = \{0, 1\}$. If cop =1 and C (node $\rightarrow$ seed), node is added to v. $dop = \{0, 1\}$, if dop =1 and D (node $\rightarrow$ seed), node is added to v. v is a subset of the node in program. Next, we slice the program with data and control flow to analyze the source code from the reading points of each configuration parameter (the program slicing strategies can be seen in Section 4.3.2). By this means, we get a set of statements that are directly affected by each configuration.

For each configuration c_j, we use $slice_j$ to denote the slicing results of c_j, taking into consideration both control and data flow. We also use $slice_{ji}$ to represent the number of statements in b_i for c_j. Then, the influence score of c_j in b_i can be calculated by $v_{ji} = \frac{slice_{ji}}{block_i}$, which indicates how the c_j influences the b_i. Finally, by calculating the influence scores for all blocks, we obtain a vector $Vector_j$ that encodes how the configuration parameter c_j influences the control and data flows of the whole program as follows.

$$Vector_j = (v_{j1}, v_{j2}, ..., v_{ji}, ..., v_{jn}) \quad i = 1, ..., n \qquad (1)$$

Despite encoding the influences of each configuration, the statements sliced from configuration parameters are not always closely related to the features of the target software. Therefore, we heuristically prune the slices to ensure that the slicing results are highly correlated with software functionalities. The pruning strategies are described as follows.

First, we only focus on the source code of the target software, ignoring the implementation details of external methods such as the library and system methods which are unrelated to configurations. Second, we prune the statements that have no effects on the data flow or control flows, such as the output statements and exceptions. Third, for function calls, we ignore the call statements and jump to the code statements which are called. Based on the above strategies, MMD is not only able to reduce the statements related to the above configuration `dfs.heartbeat.interval` from 30,000 to 5,000, but also to maintain the semantics of the vector representation. Following the example in figure 1b, the CCA module using the block structures to represent the source code as a vector. In the given example, the resultant vector has a length of 412,157. Furthermore, the CCA module constructs vectors for all configuration parameters like $Vector_{cpu\_vcores}$ and $Vector_{hardwarecapabilities}$.

3.2.3 Distance calculation

After obtaining the vector representation for each configuration parameter, we estimate the correlations among different configurations. Since the vector of each configuration reflects the functionality that the configuration is involved in, we estimate the correlation between a pair of configurations by calculating the distance between their vectors. Specifically, we calculate the distance between the configurations c_x and c_y by $Dist(Vector_x, Vector_y) = \sqrt{\sum_{i=1}^{n}(v_{xi} - v_{yi})^2}$, where V_x and V_y denote the vectors for c_x and c_y, v_{xi} and v_{yi} denote the influence scores of V_x and V_y in $block_i$.

Note that when the number of sliced statements is too small, the corresponding vector becomes too sparse and even might be a zero vector, indicating that there are no statements sliced by the seed. Similarly, given a block b_i, if both v_{xi} and v_{yi} are zeros, their distance in b_i (calculated by $v_{xi} - v_{yi}$) would be zero too. However, it does not mean that they are very close in terms of influencing the functionalities of the target software or block. For this reason, we add a bias to the Euclidean Distance by $bias = \frac{m}{n}$, where m denotes the numbers of blocks where the influence scores of both configurations are zeros, and n denotes dimension of each configuration vector (i.e., the number of blocks). Then, the distance between two configurations c_x and c_y can be calculated as follows.

$$Dist(Vector_x, Vector_y) = \sqrt{\sum_{i=1}^{n}(v_{xi} - v_{yi})^2 + bias} \qquad (2)$$

A smaller distance $Dist(Vector_x, Vector_y)$ indicates that the configurations c_x and c_y involve more common functionalities, and might be more correlated to each other. Continuing with the illustration depicted in Figure 1b, the CCA module calculates the distance between two configuration parameter vectors, like $Dist(Vector_{cpu\_vcores}, Vector_{hardwarecapabilities})$ These distances are then sorted in ascending order, yielding a static correlation result set for each configuration parameter. cpu_vcores exhibits a high rank in the correlation result set, with a distance of 0.293 from hardwarecapabilities.

3.3 Primary Misconfigurations Diagnosis

This module is based on the idea of Fault Localization [21]. First of all, the target software is executed against many test cases (TCs). A test case is a set of all value of configuration parameters. We generate the test cases through ConfErr [22], a tool to generates realistic configuration parameters. ConfErr automatically generates wrong parameters which have semantic errors and so on, randomly selects parameters to keep the default value. And information on the target software's execution is collected, which records the coverage of program statements and their status (success or failure). Then, Jaccard [23], a formula for fault location, is used to calculate the suspiciousness of each statement. Finally, MMD slices backward from the statement with the highest suspiciousness, in order to locate the configuration parameter as the root cause.

First, we randomly generate test cases in this paper. The range of configuration parameters is obtained from official user manuals, which are used to specify the value of random generation. If the range of configuration parameters is not depicted, a larger range of values will be randomly generated. The test cases are injected into the software and the collected information is represented, which is a matrix and a result vector. The value in the matrix is 1 if the test case covers the statements and 0 otherwise. The resulting vector is represented as 0 if the program runs successfully and 1 otherwise.

we get the following notations from the matrix:
$N_{cf}(s_i)$: number of failed test cases covering s_i
$N_{uf}(s_i)$: number of failed test cases not covering s_i
$N_{cs}(s_i)$: number of successful test cases covering s_i
Then, the suspiciousness of each statement is given by formula 3, and sorted in descending order of it.

$$Jaccard(s_i) = \frac{N_{cf}(s_i)}{N_{cf}(s_i) + N_{uf}(s_i) + N_{cs}(s_i)} \quad i = 1, ..., n \tag{3}$$

Finally, a slicing set is obtained by slicing backward from every suspicious statement. If the set contains read points for configuration parameters, we consider those parameters to be the root cause of the configuration error. The PMD module generates a list of configuration parameters with error rankings based on the distance between the read points and seed statements in the system dependency graph. If not, continue looking for the set of the next suspicious statements until a configuration is included. A first root cause list of suspicious configuration parameters is formed. Following the running example, during the occurrence of a configuration error, test cases are generated and injected into the software. After identifying the location of the error, the root cause configuration parameter is obtained from the slice.

3.4 Suspicious Configurations Recommendation

Getting the outputs of two modules (i.e., Correlated Configurations Analysis and Primary Misconfigurations Diagnosis), we reorder the configurations considering both their suspiciousness and correlations. Specifically, we select the correlation results from CCA module based on the results of PMD module. If the CCA module outputs the first root cause of misconfiguration, we prioritize the configuration parameters, which are the most correlated configurations for it according to the output of the PMD module. For the running example, MMD then reports the root cause along with the correlation set as an error list. In this scenario, the second configuration parameter, hardwarecapabilities, from the correlation set is identified as the cause of the error.

4 Evaluation

In this paper, we evaluate the effectiveness of MMD by answering the following questions.

- **RQ1**: How effective is MMD to diagnosis multi-misconfiguration compared with single configuration error diagnosis tool?
- **RQ2**: How do the slicing strategies affect the effectiveness of the CCA module?
- **RQ3**: How does the effectiveness of the CCA module for mining correlated configurations compare with state-of-the-art baselines?

4.1 Experiments Setup

We conduct the experiments on the a computer machine with the Intel i5-7300HQ CPU (2.5Ghz) and 16GB physical memory. We construct block structure, perform slicing and instrumentation via WALA [24], a bytecode analysis framework for Java. First, we add a forward slicing function to WALA, in order to get the statements influenced by configuration parameters. Then, the slicing results performed in CCA and PMD module can be shared, which reduces the running time of MMD and significantly improves the efficiency of it. As for pruning, Intermediate Representation (IR) is used to denote statements, which is the language of an abstract machine and is easier to character matching. In our dataset, we observed only slight differences in the suspiciousness values of statements when utilizing different formulas [25], [26], [27], [28], [29]. Furthermore, the Jaccard demonstrated a better result. What's more, to get the first root cause, the PMD module can be integrated with any formula used to derive suspiciousness scores for statements, because the process of locating the first root cause and identifying correlated configurations are two separate modules.

We use the error injection tool ConfErr [22] to randomly generate test cases and inject them into the software based on the number of profiles generated by ConfDiagnoser. For other objects not covered in ConfDiagnoser, we use the number of test cases according to the line of code. If the test cases for injection cover more sliced code, they can be more accurately located to the statement, and thus to the configuration parameters. Thus, the quality of the test case is determined by the coverage of sliced code. Table 1 shows the information of test cases. Due to the large size of Hadoop components, many functions are not covered in our tests. Therefore the code coverage is lower compared to Randoop, Soot, and Synoptic.

TABLE 1: Information of Test Cases

Program	Number of test cases	Average of code coverage
Randoop	12	92.5%
Soot	16	87.6%
Synoptic	10	91.4%
Hdfs	64	52.8%
Hbase	75	58.6%
Yarn	63	51.1%
Zookeeper	10	49.3%

Three authors manually checked the correlations inferred by CCA module, during labeling, and we took Fleiss' Kappa to assess the reliability of agreement among the raters [30]. The result is 0.83, indicating that the raters reach high agreement.

4.2 Subject Projects and Dataset

We conduct experiments on seven popular Java projects to evaluate the effectiveness of MMD, i.e., Randoop, Soot, Synoptic, Hdfs, HBase, Yarn, and ZooKeeper. These subject projects have been widely used in previous works [10], [11], [12], [15]. The statistics of the subject projects can be seen in Table 2. Since there exists no standard datasets for multi-misconfiguration diagnosis, we created the dataset from two aspects, i.e., StackOverflow [19] and error injection.

First, we collected real-world multi-misconfigurations from StackOverflow [19]. Specifically, we search issues from January 2012 to December 2022 with a set of keywords (i.e., configuration, configure, config, multi-configuration,

TABLE 2: Subject Projects and Dataset

Project	# Config	LOC	# Error	Source of errors
Randoop	57	18587	3	manually crafted
Soot	49	159273	3	manually crafted
Synoptic	37	19153	2	manually crafted
Hdfs	431	644K	5	[31], [32], [33], [34], [35]
HBase	202	755K	4	[36], [37], [38], [39]
Yarn	397	639K	4	[40], [41], [42], [43]
ZooKeeper	51	105K	1	[44]

configuration correlation, multiple configuration, and option correlation). We found 86 issues for the components of Hadoop (i.e., Hdfs, HBase, Yarn, and ZooKeeper). Then, we manually checked these issues and filtered out issues that are either irrelevant to configuration errors or only relevant to single-configuration errors. We manually reproduced the issues and maintained the cases that could be reproduced for diagnosis. Finally, we obtained 14 real-world multi-misconfiguration cases from StackOverflow.

Since there exist few relevant issues about Randoop, Soot, and Synoptic on StackOverflow, we manually crafted the multi-misconfigurations for these three projects following previous works on injecting errors into configurations [22]. Specifically, we first analyze the configuration files to obtain the configuration options. Then, We randomly select a set of correlated configurations from the dataset in a previous study [15], and manually check the presence of correlations between these configurations. We randomly change the values of correlated configuration parameters by introducing spelling, structural, and semantic errors. Finally, if the software crashes or experiences a silent error due to the change of configuration values, we add it into the multi-misconfiguration dataset.

4.3 Results

In this section, we display the experimental results, so as to evaluate the effectiveness of MMD.

4.3.1 Answers to RQ1

This section intends to evaluate the effectiveness of PMD module and MMD. PMD and MMD are both focused on diagnosing configuration errors. Specifically, PMD is used to diagnose single configuration parameter errors, while MMD is used for multi-misconfiguration errors. We evaluated the effectiveness of both methods in diagnosing these types of errors. Meanwhile, MMD is compared with an existing classic diagnosis tool named ConfDiagnoser [10]. Most of configuration error diagnosis tools output an error list with configuration parameters in order. The higher ranking of configurations, the higher suspiciousness of causing an error. Therefore, the rank of real configuration parameter in the error list that leads to an error is usually used as a criteria to judge the effectiveness of tools. The smaller this rank is (the higher it is), the better the tool is at diagnosing.

Injecting errors from the multi-misconfiguration dataset, which consists of a total of 22 errors, into target projects and simultaneously running MMD, we obtain the error list. Additionally, within our multi-misconfiguration dataset, two configuration parameters (”1st” and ”2nd”) contribute to the occurrence of multi-misconfigurations. The rank of the real configuration parameter that caused the error is summarised in Table 3. The 1st error is the rank of the actual misconfiguration parameter in the first root cause list given by PMD module, the 2nd error records the rank of the second actual configuration parameter in the error list given by MMD. ConfDiagnoser is a single configuration error diagnostic tool, so it can only diagnosis 1st error and cannot list the other suspected configuration parameters in a multi-misconfiguration. Therefore, in the 2nd error we record the rank of the second actual configuration parameter in the error list given by ConfDiagnoser that causes multi-misconfiguration.

As for the 1st error, it can be seen that MMD has an average rank of 1 for single configuration errors and ConfDiagnoser has an average rank of 1.22, which is a small difference between them. First of all, ConfDiagnoser only takes the predicates (control flow) affected by the configuration parameters into consideration, which makes it possible to reduce the content of analysis, but ignores another part of the program execution path, the data flow. Meanwhile, ConfDiagnoser requires a database for profiles in advance, and the completeness of the database limits its diagnostic effectiveness. MMD adds the data flow to the analysis process, making it more comprehensive. Furthermore, MMD performs spectrum analysis and fault location calculations by randomly generating test cases, abandoning the database comparison and making the method more practical. MMD has improved the diagnosis of single configuration errors in both aspects and has resulted in an average ranking improvement of 0.22.

When comes to the 2nd error, it can be seen that the average rank of the real configuration parameters in the correlation list given by MMD reaches 1.13. However, the average ranking of ConfDiagnoser is 7.13, which is an increase of 6 compared to MMD. The difference comes from the assumption of ConfDiagnoser, which is the error caused by single parameter. For highly suspicious statements it only adds the first configuration parameter found by DFS to the error list, and the error list is linear and does not take correlations among configuration parameters into consideration. MMD improves the rank of multi-misconfigurations by 6 through CCA module, and reduces the complexity of checking and fixing the error. Overall, MMD is very effective in diagnosing multi-misconfigurations. MMD successfully finds all the actual root causes of program errors and assists users in fixing problems.

We conducted an analysis of the “1st Parameter,” “2nd Parameter,” and “Sum” of MMD and ConfDiagnoser using the Wilcoxon signed-rank test [45] and Cliff’s delta effect size [46]. In Table 3, the “p-value” column represents the p-value obtained from the Wilcoxon signed-rank test, while the “cliffs delta” column displays the statistic for Cliff’s delta.

From the results, it is evident that the p-values for all three groups of data are less than the commonly used significance level of 0.05, indicating a significant difference between the paired samples at the 0.05 significance level. Regarding the “1st Parameter,” the p-values are comparatively larger than the other group due to close rank values in this experiment. Nevertheless, MMD still exhibits an average improvement of 0.22 compared to ConfDiagnoser.

TABLE 3: Comparison with ConfDiagnoser

Project	ID	1st Paramater		2nd Parameter		Sum	
		MMD	ConfD.	MMD	ConfD.	MMD	ConfD.
Randoop	1	1	1	1	5	2	6
	2	1	2	1	3	2	5
	3	1	1	1	4	2	5
Soot	4	1	1	1	5	2	6
	5	1	1	1	3	2	4
	6	1	2	1	5	2	7
Synoptic	7	1	1	1	4	2	5
	8	1	1	1	5	2	6
Hdfs	9	1	1	1	19	2	20
	10	1	1	1	7	2	8
	11	1	1	2	13	3	14
	12	1	2	1	15	2	17
	13	1	1	1	6	2	7
Hbase	14	1	1	1	6	2	7
	15	1	2	1	9	2	11
	16	1	1	2	4	3	5
	17	1	1	1	9	2	10
Yarn	18	1	1	2	8	3	9
	19	1	1	1	11	2	12
	20	1	2	1	7	2	9
	21	1	1	1	5	2	6
Zookeeper	22	1	1	1	4	2	5
Average		1	1.22	1.13	7.13	2.13	8.36
p-value		0.025		4.77E-07		4.77E-07	
cliffs delta		-0.227		-1.000		-1.000	

ConfD. represents ConfDiagnoser.

All three Cliff's delta data are negative, suggesting that in each case, the median of the MMD tends to be smaller than the median of the ConfDiagnoser. The second and third data points show larger effect sizes, indicating a more substantial difference between the "2nd Parameter" and "Sum" in comparison. The effect size of the "1st Parameter" is smaller, and the difference is less pronounced, but it still shows statistical significance.

For the "2nd Parameter" and "Sum," the results of Wilcoxon signed-rank test and Cliff's delta demonstrate a significant difference between MMD and ConfDiagnoser, reinforcing the superiority of MMD over ConfDiagnoser in multi-misconfiguration diagnosis.

On average, MMD takes approximately 5 minutes and 14 seconds for Randoop, Soot, and Synoptic, while the larger software projects such as Hdfs, Hbase, Yarn, and Zookeeper require an average of 88 minutes and 47 seconds. It is important to note that program analysis of larger software projects naturally takes more time due to their complexity. To mitigate the time overhead, MMD leverages result reuse, which significantly reduces the execution time of the tool by utilizing previously computed results.

Although MMD can only provide an error list without a specific fixing advice, we briefly describe how error fixing can be performed using the error list based on evaluation. For the example of Figure 6, users can modify Top-1 configuration in PMD by prioritizing, if the software runs smoothly, then a single configuration error has occurred; if the error still exists, then multi-misconfiguration is considered to have occurred and an extended part of the error list should be checked. Troubleshoot correlated configurations in turn. If the error still exists after checking all the configurations correlated with Top-1 first root cause, go back to check the following configuration in PMD list. This is a rare situation, and you can see in Table 3 the first root causes ranking first. Repeat the above steps until the error is fixed.

Answer to RQ1: On average, MMD diagnosed the multi-misconfigurations by only checking 2.13 configurations, which exhibits the superiority of MMD compared to the baseline which needs to check 8.36 configurations for multi-misconfiguration diagnosis.

4.3.2 Answers to RQ2

Our intuition for determining whether there is a correlation among configuration parameters is to calculate the distance between the vectors represented by the execution paths they affect. The closer the distance, the more similar their behaviour is likely to be, and the more likely they are to be correlated. The execution path refers to data flow and control flow, where changes in the data flow can represent the transfer of values of variables and the iterative process, and changes in the control flow can represent the jumping path of statements. We argue that both data flow and control flow can respectively detect the different types of correlations mentioned in Section 2. Therefore, in this section we focus on the impact of using three slicing strategies on the accuracy and efficiency of the correlation analysis results.

We construct vectors according to three slicing strategies, data flow, control flow and a combination of both, and then record the number of correlations under the three strategies. As can be seen in Table 4, using the data flow strategy identifies 89% of the correlations, but there are some false positives. The control flow strategy has no false positives, but the number of correlations extracted is 11%. We have analyzed the results by comparing the results and codes within the two strategies.

Control flow explicitly represents the execution path of a program through jump statements, so correlations like the one in Figure 2 will be identified easily. Such control flow correlations are relatively simple, even when jumping into other functions, only the jump statements are fetched into vector, so the false positive is very low when just using a control flow strategy. However, the number of correlations with it is low. Data flow can track the transformation of variables and it also works when configuration parameters contain iterations of values in jump statements. This is the reason why data flow strategies can get a higher number of correlations, which cause false positives. These false positives are briefly described in RQ2, and we will explain them in terms of data flow. As in Figure 3, in order to ensure a high degree of code reusability, many simple value changes among configuration parameters need to be passed as arguments to other functions. Such functions are common to many configuration parameters and can lead to false positives if the configuration parameters have a short execution path, i.e. a sparse vector.

The combination of data and control flow slicing strategy identifies more correlations, and keeps false positives to a low level by adopting a heuristic approach during the experiment. Although the slicing strategy with the two combinations takes longer to identify correlations, we need to find as many correlations as possible, and the results of CCA can be retained until the next update of program

TABLE 4: Comparison of Three Slicing Strategies

Project	Data flow				Control flow				Data flow & Control flow			
	Identify	Known TP	New TP	FP	Identify	Known TP	New TP	FP	Identify	Known TP	New TP	FP
Randoop	29	0	27	2	7	0	7	0	33	0	31	2
Soot	42	0	40	2	6	0	6	0	47	0	45	2
Synoptic	13	0	13	0	5	0	5	0	15	0	15	0
Hdfs	98	37	56	5	20	3	17	0	111	38	68	5
HBase	68	8	57	3	17	5	12	0	74	10	61	3
Yarn	182	40	130	12	34	15	19	0	203	45	146	12
ZooKeeper	22	4	17	1	5	3	2	0	27	4	22	1
Overall	454	89	340	25	94	26	68	0	510	97	388	25

without having to analyze it multiple times. Therefore, the runtime does not affect the choice of slicing strategy.

Answer to RQ2: With the data flow analysis, MMD identified 360 more configuration correlations compared to the control flow analysis, with slightly higher false positive rate. However, combining the data and control flow analysis, MMD achieved the best performances compared with other settings.

4.3.3 Answers to RQ3

We record the result of CCA module in the seven Java projects and count the total number of correlations (as shown in Table 5) for each project according to 2 configuration parameters as 1 correlation combining the data and control flow analysis. Then, manually go through the source code and user manual to check whether these correlations follow the notion of correlation described in Section 3.2 . If it does not, it is a False Positive (FP) (as shown in Table 5 FP). TP indicates the true number of correlations at the end of our manual checks. We also reproduced the previous work, cDep, as a benchmark for comparison.

As shown in Table 5, CCA module effectively identified 510 correlations in total with a low FP rate (i.e., 4.9%). CCA can be seen to be highly accurate and effective in correlation analysis. MMD's FP rate is 4.9%, which is acceptable. It is mainly due to the fact that the execution path of some configuration parameters will go through a large number of inevitable statements and methods, such as entering the same constructor declaration function and initialization function. In addition, the infrequent use of these configuration parameters makes the distance between the execution paths very close, resulting in false positives.

As Chen et al. [15] constructs a correlated dataset by manual observation in order to evaluate cDep. However, FP is not recorded according to the classification of the software, we re-run cDep and count it against the manual dataset. As Table 5 shows, the cdep has an FP rate of 15.6%, which is higher than the MMD. The precision of MMD is $precision = \frac{TP}{TP+FP} = \frac{485}{485+25} = 95.1\%$. Because of the huge amount of software code, it is difficult to construct an accurate dataset of correlations for the configuration parameters. The correlation between configuration parameters can be relatively complex. Despite the software being small in scale, its functionality complexity makes it challenging to determine the absence of correlation between the two parameters. At the same time, the number of correlations between configuration parameters is much larger than the number of configuration parameters. For instance, in the case of the Synoptic, although it only has 37 configuration parameters, the maximum number of correlations is 666. Therefore, it is also hard to count TN and FN. Using the results of cDep as a benchmark for comparison, FN is 237 in MMD, it did not miss any reports. In other words, the recall rate of MMD compared to cDep is 100%.

The reason for MMD to have a good performance in TP is that it is a statistics-based method, which allows more correlated configurations to be found than a code pattern-based method. Specifically, the code patterns defined by cDep rely on the knowledge and expertise of experts, which leads to limitations in its flexibility. An example is two configuration parameters `dfs.datatransfer.server.fixedwhitelist.file` and `dfs.datatransfer.client.fixedwhitelist.file` in Hdfs [8] (hereinafter referred to as `server` and `client`). The configuration `server` overrides the value of `client` by using object passing. This is a complex transfer relationship through other basic blocks and objects, which is difficult for cDep to discover. The configuration `server` overrides the value of `client` by using object passing, which is a complex execution path through other basic blocks, making it a mismatch with the code pattern of cDep. What's more, cDep makes efforts to prevent over-contamination are also responsible for their omissions.

In addition to the undefined code patterns, CCA module of MMD is able to find covert correlations of configuration parameters, all of which affect a certain function of the software but cannot be easily connected in source code. For example in Figure 1c, the intersection of values specified by the two configurations should be empty, but we did not find out how the two configuration parameters are related by examining the code. These correlations cannot be detected by using cDep, as it does not match any common code pattern. However, they can be discovered by the CCA module in MMD is based on a statistics method. This is mainly because the correlated configuration parameters both influence many of the same methods and statements, which means all of them work on closely related functionality in the software. MMD uses program slicing to extract the control flow and data flow of configuration parameters, which could reflect configuration parameters' behavior. The higher overlap of the same method and statements, the higher the likelihood configuration parameters work together and correlate with others. MMD effectively complements the dependency in cDep's "one-off" code pattern.

This experiment suggests that we cannot only focus on the configuration correlations at the code level, but also

TABLE 5: Results of Mining Correlated Configurations

Project	LOC	MMD		cDep	
		FP	TP	FP	TP
Randoop	18.6K	2	31	0	0
Soot	159K	2	45	0	0
Synoptic	19.1K	0	15	0	0
Hdfs	644K	5	106	26	68
Hbase	755K	3	71	6	33
Yarn	639K	12	191	10	120
Zookeeper	105K	1	26	2	16
Overall		25	485	44	237

should pay more attention to the correlations hidden in deep functionalities.

Answer to RQ3: MMD successfully identified 510 configuration correlations in seven popular Java projects with a 4.9% false positive rate, which demonstrated the superiority of MMD compared with the baseline.

5 Limitations and Threats to Validity

5.1 Limitations

The types of multi-misconfigurations diagnosed by MMD are limited. MMD provides a correlation list of each configuration parameter. However, the number of parameters that cause multi-misconfiguration in the real world is likely to be smaller than the correlation one, and there may be multi-misconfigurations that are not due to correlation at all. Therefore, MMD has false positives and false negatives by diagnosing multi-misconfigurations based on the correlation among configuration parameters.

MMD does not generate fine-grained correlation categories. Compared with cDep, MMD is more general because it does not require an additional definition of the code pattern. When encountering a wide variety of Java software, MMD can keep low false positive rate, which is not limited to pattern definition. Unfortunately, our current work is only focused on identifying whether there is a correlation among configurations, MMD cannot further specify the exact type of correlation like cDep.

MMD advises the user to fix the error according to the error list, but the process of fixing could be long. MMD outputs a list of suspicious error list which has been proven that the actual root cause ranks pretty high. However, how to fix the error is quite complex that can consume a lot of time.

5.2 Threats to Validity

The dataset of multi-misconfigurations is not large. Since it is the first work to concentrate on multi-misconfiguration, there exists no multi-misconfiguration dataset that can be used in the evaluation. Thus, we construct a dataset including 22 multi-misconfigurations for evaluating MMD. According to the definition of multi-misconfigurations, the number of configuration parameters causing errors should be variable. However, our dataset may not be large enough due to the manual cost of reproducing multi-configuration errors. In addition, we only take into consideration 2-configuration parameter-errors, ignoring the errors caused by more then two configurations, which can be rarely seen in the real world.

A small part of the misconfiguration dataset is not from the real world. As we mentioned in Section 4, we collected part of dataset from StackOverflow. In order to compare with previous work, we have to take Randoop, Soot and Synoptic into consideration. However, due to their infrequent use of them, we did not find any multi-misconfiguration issues about them. Therefore, manually crafted multi-misconfigurations were added to dataset as mentioned in Section 4.2, making part of our result unrepresentative in the real world. At the same time, the multi-misconfiguration dataset were obtained through manual observation and verification, making it difficult to find all indirect correlations, as shown in Fig. 1c. Consequently, the experiments may not effectively capture indirect configuration correlations in Randoop, Soot, and Synoptic.

The benchmark of correlation dataset may be not complete. The one and only recognized and complete dataset of correlation is cDep, which includes Hadoop only. To find out if MMD could detect correlations among configuration parameters as many as possible, we constructed a benchmark for Randoop, Soot and Synoptic by checking the user manual, which inevitably results in false negative of this benchmark.

6 Related Work

Current research in the area of configuration errors includes error diagnosis, error injection [22], [47], and repairing [48], [49], [50]. The idea of error injection is to generate a large number of test cases that violate the constraints of configuration parameters, then inject them into the software to detect whether the source code is robust. The repairing realizes how to convert from the wrong to the right state of the software after locating the error configuration. Error diagnosis is divided into the white- and black-box-based approaches, depending on whether software source code is used.

6.1 Black-box Configuration Diagnosis

The core method of black-box-based diagnosis is the discovery of constraint rules or repeated trial-and-error on the historical data and auxiliary files after the software has been run for diagnosis. Specifically, Yuan et al. [51] identify sequences of events in the Windows registry, generate sequences of contextual rules, detect the sequence of execution at runtime, then identify and filter the exceptions that occur. Encore [52] considers the interaction between configuration settings and the execution environment to learn configuration rules. Talwadker et al. [53] implement the Dexter, a tool that uses storage system logs for problem detection, which ranks log messages and matches keywords by heuristics, and later provides solutions to problems by uncovering relevant system commands and execution logs. PracExtractor [54] manages and detects configuration parameters by performing natural language processing on the text information in the user manual, extracting relevant content about the configuration information in the

user manual, and organizing it into constraints. black-box-based diagnosis focuses on the external output results of the program and it does not observe how applications use configuration parameters, so its accuracy is lower than that of White-box-based diagnosis.

6.2 White-box Configuration Diagnosis

The white-box-based diagnosis uses the software source code or binary code to analyze and combine statistical analysis, replay techniques, dynamic staking, etc., to find the configurations that may be wrong. Specifically, Sherlog [55] uses information from runtime logs to analyze source code and infer information about the execution path of the program at the time of the configuration error. This approach requires logs to be complete and accurate, and the effectiveness of log analysis relies on the developer's domain knowledge, which makes this approach highly uncertain. confAid [56] performs dynamic information flow analysis using binaries and configuration files. ConfDebugger [12] and ConfDoctor [13] diagnose configuration errors by taking the intersection of the forward slicing of configuration parameters and the backward slicing of program statements in the stack trace. They all focus exclusively on configuration errors that lead to a crash or assertion failure. However, MMD can diagnose any type of configuration error. ConfDiagnoer [10] compares the correct and incorrect execution profiles, finds predicates on the incorrect execution path, and finally matches them to the incorrect configuration parameters. ConfSuggester [11] is a new application of ConfDiagnoer for diagnosing configurations that have changed since the version update. The two work above diagnose both crash and non-crash configuration errors, like all White-box-based diagnosis, they assume that only one configuration parameter causes the error, whereas MMD can address multi-misconfigurations.

Configuration parameter representation in code. Few parts of the researches in program analysis-based diagnostics focus on the representation in configuration parameters. SPEX [57] classifies the types of configuration parameters, traces the data flow with configuration parameters, and automatically records inferences of constraint rules for individual configuration parameters under that path. The constraint rules here refer to individual configuration parameters. ConfigX [58] and cDep [15] concentrate on the representation among configurations. ConfigX automatically infers rules based on human definitions, and cDep detects dependencies among configuration parameters by pattern matching and taint propagation. Both of them require pre-definition, which is not fully automated and the results are dependent on the accuracy and completeness of the pre-definition. Microsoft designed Rex [59], a tool, using machine learning and program analysis, learns change-rules to capture correlations. In order to scale well, Rex mine the correlations at the file-level.

6.3 Configurations in Software Product Lines

The term of configuration is also used in the field of software product lines (SPLs) [60], [61], but it is different from the configuration in this paper. In SPL, a feature refers to a functionality of a software product that meets a requirement [62]. By building a tree hierarchy, the interactions between features can be modeled and analyzed, based on which configuration analysis (also mentioned as variability analysis [63]) is conducted to combine features in SPLs [64]. However, in this paper, a configuration refers to a key-value pair in source code that can be configured by users, which is not only a switch of a functionality, but also other types of settings (e.g., `diskCheckTimeout` in Figure 2).

Jens et al. [65] analyze the differences between feature flags and configuration options. Specifically, feature flags are developer-oriented and need to be deleted when the development process is completed, while configuration options are user-oriented and maintained in software. Variability bugs in SPLs refer to bugs that involves at least one feature that have to be enabled or disabled to make the bug occur [66]. However, in the field of misconfiguration diagnosis, a configuration error (i.e., misconfiguration) refers to incorrectly setting of one or multiple configuration options (e.g., the value of a configuration is too large). To sum up, although the terms in SPLs and this paper might have some overlap, their definitions are different. As previous studies in the literature [9], [10], [12], [13], [22], [53], [58], this paper focuses on diagnosing the errors of configuration parameters at the source-code level, which are not only a switch of a functionality.

7 Conclusion and Future Work

This paper presents MMD, a tool for diagnosing errors, which are caused by multiple configuration parameters of the software. It is based on program statistics analysis and takes software source code as input. MMD generates error lists by combining a list of configurations ranked by suspiciousness and correlated configuration parameters to help users diagnose multi-configurations. Our evaluations show that it can effectively diagnose real-world multi-misconfiguration with low false positive rate, and can find more cases than other state-of-the-art work. In the future, we could expand our system by adopting more effective program analysis frameworks and apply the idea to other software in different languages. In the future, we will expand our system by adopting more effective program analysis frameworks, applying the idea to other software in different languages, and building a configuration parameter correlation library for more software to assist developers to inject multi-misconfigurations for adding robustness. And generate a true positive dataset of correlated configurations in this research area.

8 Acknowledgments

The authors would like to thank the Editor and the reviewers for their constructive comments and valuable feedback. This work is supported in part by the Fundamental Research Funds for the Central Universities of China under Grant 2020JBZ104, the National Natural Science Foundation of China under Grant U21A20463, U22B2027, 62172297, 62202245, 62102198, 61971029, 61902276, Tianjin Intelligent Manufacturing Special Fund Project under Grants 20211097, and China Postdoctoral Science Foundation (No. 2021M691673, No. 2023T160335).

REFERENCES

[1] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo, "Software configuration engineering in practice interviews, survey, and systematic literature review," *IEEE Transactions on Software Engineering*, vol. 46, no. 6, pp. 646–673, 2018. (document)

[2] Apache hadoop. [Online]. Available: http://hadoop.apache.org (document)

[3] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 159–172. (document)

[4] A. Rabkin and R. H. Katz, "How hadoop clusters break," *IEEE software*, vol. 30, no. 4, pp. 88–94, 2012. (document)

[5] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic configuration management at facebook," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 328–343. (document)

[6] Jchord properties. [Online]. Available: https://www.seas.upenn.edu/~mhnaik/chord/user_guide/properties.html (document)

[7] Yarn. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-yarn (document)

[8] Hdfs. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html (document), 3.2, 4.3.3

[9] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 193–202. (document), 6.3

[10] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 312–321. (document), 2.2, 4.2, 4.3.1, 6.2, 6.3

[11] ——, "Which configuration option should i change?" in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 152–163. (document), 4.2, 6.2

[12] Z. Dong, M. Ghanavati, and A. Andrzejak, "Automated diagnosis of software misconfigurations based on static analysis," in *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2013, pp. 162–168. (document), 2.2, 4.2, 6.2, 6.3

[13] Z. Dong, A. Andrzejak, and K. Shao, "Practical and accurate pinpointing of configuration errors using static analysis," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 171–180. (document), 2.2, 6.2, 6.3

[14] S. Mehta, R. Bhagwan, R. Kumar, C. Bansal, C. Maddila, B. Ashok, S. Asthana, C. Bird, and A. Kumar, "Rex: Preventing bugs and misconfiguration in large services using correlated change analysis," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 435–448. (document)

[15] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and discovering software configuration dependencies in cloud and datacenter systems," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 362–374. (document), 2.1, 2.2, 4.2, 4.3.3, 6.2

[16] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816. (document)

[17] Soot. [Online]. Available: https://github.com/soot-oss/soot (document)

[18] Synoptic. [Online]. Available: https://github.com/modelinference/synoptic (document)

[19] Stackoverflow. [Online]. Available: https://stackoverflow.com (document), 4.2

[20] B. Randell and J. Xu, "The evolution of the recovery block concept," *Software fault tolerance*, vol. 3, pp. 1–22, 1995. 3.2.1

[21] B. Liu, Lucia, S. Nejati, L. C. Briand, and T. Bruckmann, "Simulink fault localization: an iterative statistical debugging approach," *Software Testing, Verification and Reliability*, vol. 26, no. 6, pp. 431–459, 2016. 3.3

[22] L. Keller, P. Upadhyaya, and G. Candea, "Conferr: A tool for assessing resilience to human configuration errors," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 157–166. 3.3, 4.1, 4.2, 6, 6.3

[23] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, "Using of jaccard coefficient for keywords similarity," in *Proceedings of the international multiconference of engineers and computer scientists*, vol. 1, no. 6, 2013, pp. 380–384. 3.3

[24] Wala. [Online]. Available: http://sourceforge.net/projects/wala 4.1

[25] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013. 4.1

[26] P. Arumuga Nainar and B. Liblit, "Adaptive bug isolation," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 255–264. 4.1

[27] E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *2008 1st international conference on software testing, verification, and validation*. IEEE, 2008, pp. 42–51. 4.1

[28] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 2002, pp. 467–477. 4.1

[29] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98. 4.1

[30] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971. 4.1

[31] misconfiguration 9. [Online]. Available: https://stackoverflow.com/questions/48354481/namenode-doesnt-detect-datanodes-failure/48354911#48354911 **??**

[32] misconfiguration 10. [Online]. Available: https://stackoverflow.com/questions/70419926/why-does-a-datanode-doesn **??**

[33] misconfiguration 11. [Online]. Available: https://stackoverflow.com/questions/37638558/is-there-any-file-to-read-when-on-of-my-datanode-was-dead **??**

[34] misconfiguration 12. [Online]. Available: https://stackoverflow.com/questions/29712585/datanode-decommisioning-vs-write/29714047#29714047 **??**

[35] misconfiguration 13. [Online]. Available: https://stackoverflow.com/questions/23878557/is-it-possible-to-replicate-the-namenode-in-hadoop **??**

[36] misconfiguration 14. [Online]. Available: https://stackoverflow.com/questions/7176254/not-able-to-connect-to-remote-hbase/11518026#11518026 **??**

[37] misconfiguration 15. [Online]. Available: https://stackoverflow.com/questions/22528859/hbase-scan-performance/22547731#22547731 **??**

[38] misconfiguration 16. [Online]. Available: https://stackoverflow.com/questions/28869180/on-master-node-failed-construction-of-regionserver-java-net-bindexception/30975132#30975132 **??**

[39] misconfiguration 17. [Online]. Available: https://stackoverflow.com/questions/30923351/hbase-client-rpc-timeout/44684771#44684771 **??**

[40] misconfiguration 18. [Online]. Available: https://stackoverflow.com/questions/38946415/how-to-decrease-heartbeat-time-of-slave-nodes-in-hadoop/39012675#39012675 **??**

[41] misconfiguration 19. [Online]. Available: https://stackoverflow.com/questions/22944853/map-reduce-jobs-failed-with-virtual-memory-beyond-limit-in-yarn-cluster/45928641#45928641 **??**

[42] misconfiguration 20. [Online]. Available: https://stackoverflow.com/questions/59863850/how-to-control-the-number-of-hadoop-ipc-retry-attempts-for-a-spark-job-submissio/60011708#60011708 **??**

[43] misconfiguration 21. [Online]. Available: https://stackoverflow.com/questions/29940711/apache-spark-setting-executor-instances-does-not-change-the-executors **??**

[44] misconfiguration 22. [Online]. Available: https://stackoverflow.com/questions/44587945/unable-to-start-zookeeper-server **??**

[45] R. F. Woolson, "Wilcoxon signed-rank test," *Wiley encyclopedia of clinical trials*, pp. 1–3, 2007. 4.3.1

[46] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, "Cliff's delta calculator: A non-parametric effect size program for two groups of observations," *Universitas Psychologica*, vol. 10, no. 2, pp. 545–555, 2011. 4.3.1

[47] S. Li, W. Li, X. Liao, S. Peng, S. Zhou, Z. Jia, and T. Wang, "Confvd: System reactions analysis and evaluation through misconfigura-

tion injection," *IEEE Transactions on Reliability*, vol. 67, no. 4, pp. 1393–1405, 2018. 6

[48] A. Mahgoub, P. Wood, A. Medoff, S. Mitra, F. Meyer, S. Chaterji, and S. Bagchi, "{SOPHIA}: Online reconfiguration of clustered {NoSQL} databases for {Time-Varying} workloads," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 223–240. 6

[49] C.-J. Hsu, V. Nair, T. Menzies, and V. W. Freeh, "Scout: An experienced guide to find the best cloud configuration," *arXiv preprint arXiv:1803.01296*, 2018. 6

[50] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki, "Range fixes: Interactive error resolution for software configuration," *Ieee transactions on software engineering*, vol. 41, no. 6, pp. 603–619, 2014. 6

[51] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, 2011, pp. 28–28. 6.1

[52] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014, pp. 687–700. 6.1

[53] R. Talwadker, "Dexter: faster troubleshooting of misconfiguration cases using system logs," in *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017, pp. 1–12. 6.1, 6.3

[54] C. Xiang, H. Huang, A. Yoo, Y. Zhou, and S. Pasupathy, "Pracextractor: Extracting configuration good practices from manuals to detect server misconfigurations," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 265–280. 6.1

[55] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from run-time logs," in *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, 2010, pp. 143–154. 6.2

[56] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis." in *OSDI*, vol. 10, no. 2010, 2010, pp. 1–14. 6.2

[57] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 244–259. 6.2

[58] J. Zhang, R. Piskac, E. Zhai, and T. Xu, "Static detection of silent misconfigurations with deep interaction analysis." *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–30, 2021. 6.2, 6.3

[59] S. Mehta, R. Bhagwan, R. Kumar, C. Bansal, C. Maddila, B. Ashok, S. Asthana, C. Bird, and A. Kumar, "Rex: Preventing bugs and misconfiguration in large services using correlated change analysis," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 435–448. 6.2

[60] D. M. Weiss and C. T. R. Lai, *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., 1999. 6.3

[61] J. M. Horcas, D. Struber, A. Burdusel, J. Martinez, and S. Zschaler, "We're not gonna break it! consistency-preserving operators for efficient product line configuration," *IEEE Transactions on Software Engineering*, 2022. 6.3

[62] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans, "Feature model extraction from large collections of informal product descriptions," in *proceedings of the 2013 9th joint meeting on foundations of software engineering*, 2013, pp. 290–300. 6.3

[63] A. V. Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel, "Variability-aware static analysis at scale: An empirical study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 4, pp. 1–33, 2018. 6.3

[64] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Computing Surveys (CSUR)*, vol. 47, no. 1, pp. 1–45, 2014. 6.3

[65] J. Meinicke, C.-P. Wong, B. Vasilescu, and C. Kästner, "Exploring differences and commonalities between feature flags and configuration options," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, 2020, pp. 233–242. 6.3

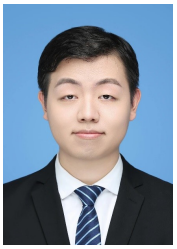
[66] I. Abal, J. Melo, c. Stănciulescu, C. Brabrand, M. Ribeiro, and A. Wasowski, "Variability bugs in highly configurable systems: A qualitative analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 3, jan 2018. [Online]. Available: https://doi.org/10.1145/3149119 6.3

Yingnan Zhou is a master's student at the College of Computer Information and Technology, Beijing Jiaotong University, China, major in computer technology. She was born in Zibo, Shandong Province, China, in 1998. She graduated from Inner Mongolia University with a bachelor's degree in Computer Science and Technology in 2021. Her main research direction is configuration error diagnosis. She is interested in configuration security, program analysis and software security.

Xue Hu is a master's student at the College of Intelligence and Computing, Tianjin University,China. She graduated from Fuzhou University with a bachelor's degree in Internet of Things Engineering in 2020. Her main research direction is cyberspace security. She is interested in software security, program language and artificial intelligence.

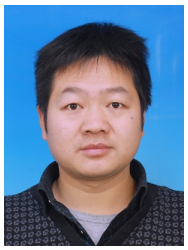
Sihan Xu received the B.Sc. and Ph.D. degrees in computer science from Nankai University in 2013 and 2018, respectively. For her research, she spent a year with the National Uni- versity of Singapore. She is currently a research fellow with College of Cyber Science, Nankai University. Her research interests include software engineering and AI security.

Yan Jia received the Ph.D. degree from the School of Cyber Engineering, Xidian University, Xi'an, China, in 2020. He is a Research Associate with the College of Cyber Science, Nankai University, Tianjin, China. His interests include discovering and understanding new design or logic security vulnerabilities in real-world systems, including IoT, Web/browser, mobile, and network. His work helped many high-profile vendors improve their products' security, including Amazon, Microsoft, Apple, and Google.



Yuhao Liu is a master's student at the School of Computer and Information Technology, Beijing Jiaotong University, China. He was born in Wuhan, Hubei Province, China, in 1998. He graduated from Xi'an University of Posts & Telecommunications with a bachelor's degree in Network Engineering in 2020. He majors in cyberspace security. He is interested in programming language, reverse engineering and software security.

Junyong Wang is a master's student at the College of Computer Information and Technology, Beijing Jiaotong University,China. He was born in Xinyang, Henan Province, China, in 1998. He graduated from Guizhou University with a bachelor's degree in Computer Science and Technology in 2020. His main research direction is cyberspace security. He is interested in software security, program language and artificial intelligence.



Guangquan Xu is a Ph.D. and full professor at the Tianjin Key Laboratory of Advanced Networking (TANK), College of Intelligence and Computing, Tianjin University, China. He received his Ph.D. degree from Tianjin University in March 2008. He is a member of the CCF and IEEE. His research interests include cyber security and trust management.

Wei Wang is a full Professor with school of computer and information technology, Beijing Jiaotong University, China. He received the Ph.D. degree from Xi'an Jiaotong University, in 2006. He was a Post-Doctoral Researcher with the University of Trento, Italy, from 2005 to 2006. He was a Post-Doctoral Researcher with TELECOM Bretagne and with INRIA, France, from 2007 to 2008. He was also a European ERCIM Fellow with the Norwegian University of Science and Technology (NTNU), Norway, and with the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, from 2009 to 2011. He has authored or co-authored over 100 peer-reviewed articles in various journals and international conferences. His recent research interests lie in data security and privacy-preserving computation. He is an Elsevier "highly cited Chinese Researchers". He is an Editorial Board Member of Computers & Security and a Young AE of Frontiers of Computer Science.

Shaoying Liu (Fellow, IEEE) received the Ph.D. degree in computer science from The University of Manchester, U.K. He is currently a Professor of software engineering in the Dependable Systems Laboratory, Graduate School of Advanced Science and Engineering, School of Informatics and Data Science, Hiroshima University, Japan. He has developed the SOFL specification language and the related Agile-SOFL method for developing dependable systems. He has authored a book entitled Formal Engineering for Industrial Software Development: Using the SOFL Method (Springer, 2004) and over 280 refereed publications in journals and international conferences. His research interests include software engineering, formal engineering methods, software testing and verification, human-machine pair programming, and trustworthy systems. He is also a BCS Fellow, AAIA Fellow, and a member of IPSJ and IEICE.

Thar Baker (Senior Member, IEEE) (Senior Fellow, HEA) is Professor of Industry 4.0/5.0 in the School of Architecture, Technology and Engineering at the University of Brighton (UoB) in the UK. He is the Chair of Board of Governors of eSystems Engineering Society (eSES). He was the Head of Applied Computing Research Group (ACRG) in the Faculty of Engineering and Technology at Liverpool John Moores University (LJMU) in the UK. He has published numerous refereed research papers in multidisciplinary research areas, including cloud and fog computing, edge AI, IoT, sensor networks, and federated learning.