Detecting Missing-Permission-Check Vulnerabilities in Distributed Cloud Systems

Jie Lu
lujie@ict.ac.cn
SKLP, Institute of Computing Technology, CAS

Haofeng Li
lihaofeng19b@ict.ac.cn
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences, China

Chen Liu
liuchen17z@ict.ac.cn
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences, China

Lian Li*†
lianli@ict.ac.cn
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences, China

Kun Cheng
chengkun12@huawei.com
Huawei Technologies Co. Ltd, China

ABSTRACT

Missing- Permission-Check (MPC) vulnerability is a type of bug where permission checks are not enforced for privileged operations. MPC vulnerability is prevalent and can cause severe security impacts. This paper proposes the first tool to detect MPC vulnerabilities in distributed cloud systems. We conduct an in-depth study of 95 real-world MPC vulnerabilities and our findings motivate a new tool named MPCHECKER. The tool introduces a combined log-static analysis to automatically identify privileged operations by inferring variables representing user owned data and critical system states, whose accesses need to be protected. We have evaluated MPCHECKER with 6 popular distributed systems. The tool reports 44 new vulnerabilities, and 43 of them have been confirmed and labeled as *critical* bugs. Moreover, 1 bug is particular dangerous and the developers requested to keep it undisclosed.

CCS CONCEPTS

• **Security and privacy** → **Distributed systems security**.

KEYWORDS

Missing Permission Check Vulnerabilities; Distributed System

ACM Reference Format:
Jie Lu, Haofeng Li, Chen Liu, Lian Li, and Kun Cheng. 2022. Detecting Missing-Permission-Check Vulnerabilities in Distributed Cloud Systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3560589

*corresponding author: lianli@ict.ac.cn
†Also with TianQi Soft Inc., China.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3560589

1 INTRODUCTION

Distributed cloud systems are facing great security challenges. The great amount of valuable data hosted on cloud platforms have naturally caught the attention of attackers. Although various security mechanisms have been introduced to secure cloud systems, there are widely existing vulnerabilities which may be exploitable, leading to numerous cloud breaches [1, 30, 34, 52, 58].

Among all types of vulnerabilities in distributed cloud systems, access control vulnerabilities are one of the most prevalent and troublesome [11, 23, 33, 55]. Distributed cloud systems consist of multiple components, where each component provides a set of remotely accessible APIs for communication with users or with other components in the system. Very often, developers forget to introduce proper permission checks in those APIs, resulting in easily exploitable Missing-Permission-Check (MPC) vulnerabilities. As studied in [4], insecure interfaces and APIs account for 42% of total cloud security vulnerabilities. Furthermore, our study over 152 real-world access control vulnerabilities in distributed cloud systems points out that 62.5% of them are MPC vulnerabilities in public APIs. Such vulnerabilities are the direct cause of many cloud breaches [1, 13, 17, 30, 31, 34, 47–49, 57, 58, 61], such as the data leak of hundreds of thousands of booking reservations in Autoclerk, a hotel reservation management system [47].

Figure 1 gives a MPC vulnerability example from the distributed file system HDFS [12]. The function `deleteBlockPool` is a remotely invocable API (via remote procedural call, i.e., RPC). The API, as suggested by its name, will delete critical system data (line 7) and it is supposed to be accessible to the administrator only. However, there exists a MPC vulnerability in the original implementation and an attacker can invoke the API remotely to exploit this vulnerability and delete arbitrary system data, causing severe damages to the system. To fix this vulnerability, developers simply introduce a permission check at line 2, ensuring that the API is accessible to the administrator only.

```
1  public void deleteBlockPool(BlockPoolId blockPoolId,
   ↪  boolean force, String remoteUser) {
2    + checkSuperuserPrivilege(remoteUser); //permission
     ↪  check
3    LOG.info("delete block pool {} by user {}",
     ↪  blockPoolId, remoteUser);
4    if (this.blockPoolManager.get(blockPoolId) == null) {
5        throw new IOException("...");
6    }
7    this.data.deleteBlockPool(blockPoolId, force);
     ↪  //privileged operation
8  }
9
10 public void checkSuperuserPrivilege(String remoteUser) {
11   if (!remoteUser.equals(this.superUser)){
12      throw new AccessControlException();
13   }
14 }
```

Figure 1: A MPC vulnerability in HDFS (CVE-2014-0229).

1.1 Existing Efforts

MPC vulnerabilities are detected by finding those program paths from program entry (entry of the main function or entry of a publicly accessible API) to a privileged operation without permission checks. There are 2 steps involved, as follows.

1. Identify permission checks and privileged operations.
2. Check whether privileged operations are guarded by permission checks along all program paths.

For the example in Figure 1, line 7 (invocation to method `data.deleteBlockPool()`) is a privileged operation and the vulnerability surfaces on the program paths from line 1 (entry of the publicly accessible API) to the privileged operation at line 7. In the fix, a permission check (`checkSuperuserPrivilege()`) is introduced at line 2 to guard the privileged operation at line 7.

There have been a few approaches to detect MPC vulnerabilities in operating systems [56, 65], and in web applications [19, 54]. In those approaches, permission checks are manually specified. Since their targeted systems commonly implement permission checking APIs in one module (e.g., LSM in Linux [62]), the amount of manual efforts is insignificant. Given the set of manually specified permission checks, those approaches automatically mine privileged operations based on the assumption that privileged operations must have been guarded by permission checks on the majority of program paths. Thus, methods whose call-sites are frequently guarded by permission checks are considered as privileged functions and their call-sites are privileged operations. Next, after identifying permission checks and privileged operations, classic inter-procedural dataflow analysis [26, 40, 51] can be applied to find those vulnerable program paths with unguarded privileged operations. Such consistency-checking approaches [21] work well for those privileged operations that are partially protected, i.e., privileged functions whose call-sites are guarded only on a subset of program paths.

1.2 Challenges in Distributed Systems

Existing approaches, although well suitable for their targeted systems, may not be applicable to distributed systems. To understand MPC vulnerabilities in distributed systems, we conduct an empirical study of 152 real-world access control vulnerabilities from 10 widely used distributed systems. Among the 152 vulnerabilities, 95 of them (62.5%) are MPC vulnerabilities. To the best of our knowledge, this is the first in-depth study targeting access control vulnerabilities in distributed systems. Some of our findings are summarized below.

1. Different components in distributed systems often implement distinct permission checking mechanisms. Hence, it requires large amount of manual efforts to examine and specify all permission checking methods in distributed systems. In addition, 32.6% of permission checks are done ad hocly by directly testing user variables in conditional statements (e.g., line 11 in Figure 1). It is unclear how to specify those conditional statements. If permission checks could not be identified correctly, privileged operations cannot be precisely mined, making existing approaches ineffective. Moreover, a large percentage (60%) of MPC vulnerabilities are triggered by privileged operations not guarded anywhere in the program. As a result, existing mining-based approaches fail to recognize those operations.
2. In distributed systems, permission checks and privileged operations often locate in different components. Thus, it is critical to understand and precisely analyze inter-component interactions – a well known challenge in analyzing distributed systems. Existing static analyses do not understand such communication mechanisms (implemented via RPCs, message events, or web sockets), resulting in incomplete call graphs and imprecise analysis results. Furthermore, it is challenging to scale static analyses to the whole system (e.g., analyze all components of a distributed system together), given the large combined program size of all different components of a system.

The above findings highlight that existing approaches are not applicable to distributed systems, and we need new techniques to address the above challenges, and effectively detect MPC vulnerabilities in distributed systems.

1.3 Solution and Contributions

We propose a new approach to detect MPC vulnerabilities in distributed systems. Our approach automatically identifies permission checks and privileged operations in distributed systems, by inferring variables representing user owned data and critical system states. Accesses to user owned data and critical system states are privileged operations because those accesses may corrupt user data and system state, or leak insensitive information. To effectively detect MPC vulnerabilities, we formulate it as a classic inter-procedural finite subset dataflow problem (IFDS). For efficiency, instead of applying IFDS directly to the whole system, we summarize inter-component communications and apply IFDS to each component separately.

Our approach is based on the following two observations.

- Permission checks in distributed systems are performed by checking variables representing users, either directly in conditional statements (e.g., line 11 in Figure 1), or wrapped as APIs (e.g., `checkSuperuserPrivilege()` in Figure 1).
- Accesses to user owned data and critical system states are privileged operations, which need to be guarded by permission checks.

Intuitively, accesses to data owned by a user need to be checked by testing the accessing user, to avoid unauthorized accesses. In addition, only the super user can write to critical system states. Therefore, we automatically identify privileged operations by inferring variables representing user-owned data and critical system states. Those variables are referred to as *user-related* variables and *system-related* variables, respectively. The key question is: how to find user-related and system-related variables automatically?

Our study indicates that most user-related variables (90%) are printed in log statements. Hence, we apply log analysis together with static analysis to infer user-related variables. Specifically, given the set of all user names, we check runtime logs to discover variables printing the same name as a user. A type-based static analysis is followed to discover more user-representing variables in the program. Variables representing user owned data are discovered via a correlational analysis, which examines those variables printed in a same log instance with a user variable. On the other hand, system-related variables are discovered by analyzing instance fields checked as exception or error conditions (e.g., `if (this.field) abort();`). These variables are regarded as critical state variables, since their values indicate abnormal error conditions.

We realize our approach in a new tool named MPChecker and evaluate it on 6 popular distributed systems. The tool correctly discovers hundreds of privileged operations and permission checks in those systems, and reports 44 new vulnerabilities that have never been found before. To date, 20 reported bugs have been fixed and 23 bugs have been confirmed by the original developers. All confirmed bugs are labeled as *critical* bugs that may be easily exploited. Moreover, 1 bug is particular dangerous and the original developers requested to keep it undisclosed. The tool also reports 7 false positives, with a false positive rate of 13.7%.

The contributions of this paper are summarized as follows:

- For the first time, we conduct an in-depth study on 95 MPC vulnerabilities in distributed systems. Our study sheds lights on new detection techniques for MPC vulnerabilities in distributed systems.
- We propose a new log-based analysis to detect MPC vulnerabilities in distributed systems. Our approach automatically discovers permission checks and privileged operations at high precision, by inferring variables representing user owned data and critical system states.
- We develop MPChecker, the first MPC vulnerability detection tool for distributed systems. The tool successfully reports 44 new critical bugs in 6 real-world distributed systems, with a false positive rate of 13.7%.

In the rest of the paper, Section 2 studies 95 real MPC vulnerabilities in detail. We illustrate our approach in Section 3 and evaluate it in Section 4. Section 5 reviews related work and Section 6 concludes the paper.

Table 1: Distributed systems under study.

Systems	Type	Checking Mechanisms	# of bugs
HDFS[12]	Distributed file system	M1,M3,M4	8
MapReduce[20]	Data process framework	M1,M3	1
YARN[59]	Resource manager system	M1,M3	13
HBase[60]	Key value database	M1,M4	35
ZooKeeper[29]	Coordination service	M1	1
Kafka[35]	Data process framework	M1,M4	0
Mesos[27]	Resource manager system	M1,M4	25
MongoDB[16]	Document database	M1,M2	1
Kubernetes[10]	Resource manager system	M1,M2,M3,M4	3
OpenStack[50]	Cloud computing platform	M1,M3	8
TOTAL	—	—	95

2 CHARACTERISTICS STUDY

We study 10 widely used distributed systems in Table 1. Those systems provide different services (Column 2) and are written in different languages including Java, C++, Go, Scala, and Python. Our study results are publicly available at https://github.com/lujiefsi/MPChecker.

2.1 Vulnerabilities Collection

We compile a set of 152 access control vulnerabilities (including 95 MPC vulnerabilities) in the 10 systems in Table 1. For each vulnerability, we collect its description, fixing patches, and related discussions from developers. There are 49 vulnerabilities collected from the CVE (the common vulnerabilities and exposures) vulnerability database [5]. We examined all reported 368 CVEs of the 10 systems from 2012, and filtered out 49 access control vulnerabilities.

The rest 103 vulnerabilities come from the issue tracking systems of the systems under study. We query each issue tracking system using the following keywords: "*access control*", "*authorization*", "*security check*", "*permission check*", *"privilege check*", "*admin only*", and *"superuser only*". The keyword-based search returns a total of 3,058 issues. We manually checked each issue, and obtained 103 real exploitable access control vulnerabilities.

Among the 152 access control vulnerabilities, 95 (62.5%) of them are MPC Vulnerabilities which are fixed by introducing extra permission checks. All our studied systems, except Kafka, suffer from MPC vulnerabilities. Kafka transparently authorize external requests with API hooks, effectively preventing such flaws. The remaining 57 access control vulnerabilities are caused by buggy permission check implementations (e.g., checking against a wrong user, erroneous access control policies, etc). How to detect those access control vulnerabilities is worth of separate investigation.

2.2 Caveats

The 95 Vulnerabilities under study, although extensive, are by no means complete and some findings may not be generalizable. To reduce this threat, we chose a wide range of 10 popular distributed systems. Those systems range from distributed file systems, coordination services, resource management systems, key-value databases, document-oriented database, data processing systems, and cloud

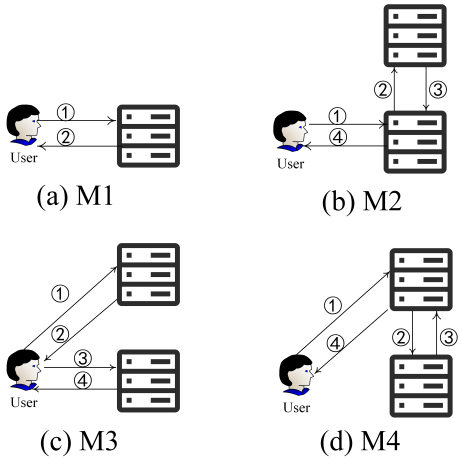

Figure 2: Four different permission checking mechanisms.

computing platforms, covering most key elements in cloud computing. The 10 systems are written in different languages (C/C++, Java, Scala, Go, and Python), and often implement distinct permission check mechanisms.

Due to the complex nature of the work and the large amount of manual efforts involved, subjectivity may exist. To reduce this threat, each vulnerability is inspected and cross-validated by at least two authors of this paper, as in previous work [7, 22].

2.3 Findings

We study each MPC vulnerability to understand its permission checking mechanisms, triggering privileged operations, and involved components. All vulnerabilities are fixed by introducing a permission check in the remotely accessible API exposing the bug.

2.3.1 Permission Checking Mechanisms. Figure 2 depicts the 4 permission checking mechanisms in distributed systems, which differ from each other on how permission checking APIs are invoked.

M1. This is the traditional mechanism where external request and permission checking are performed on the same node: ① user sends a request to the server node, ② the server node invokes corresponding permission checking APIs before returning the result of the request.

M2. In this mechanism, permission checks are performed on a separate node: ① user sends a request to the server node, ② the server node sends a permission check query to the authorization node (via RPC or web socket), ③ the server node receives response (accept or deny) from the authorization node, ④ if permission granted, the server node responses to the request and returns the result.

M3. This is token-based authorization: ① user asks for a token from the authentication node, ② a token with authentication and authorization information is returned, ③ user sends a request together with authorization token to the server node. ④ the server node checks permission against the token and the user, before responding the request.

```
1  private Set<Application> getApps(String remoteUser){
2      Set<Application> result = new HashSet<Application>();
3      for (Application app:this.applications){
4      +  if(!app.getUser().equals(remoteUser)){ //ad-hoc
       ↪   permission check
5      +     continue;
6      +  }
7         result.add(app); // privileged operation
8      }
9      return result;
10 }
```

Figure 3: An ad hoc permission check example (YARN-8455).

M4. This mechanism employs a centralized node to authorize and dispatch user requests: ① user request is handled by the authorization node, ② if permitted, the authorization node forwards the request to the server node, ③ the server node handles the request and returns to the authorization node, ④ the authorization node forwards the response to the user.

As shown in Table 1 (Column 3), each system under study adopts one or more mechanisms, and the classic mechanism M1 is implemented in all studied systems. M2 performs permission check on a remote node, while the other 3 mechanisms check permission locally via direct API calls.

Although being invoked differently in different mechanisms, all permission checking APIs implement permission checks in a similar fashion – by testing user variables. For instance, in our example in Figure 1, the API `checkSuperuserPrivilege()` simply tests whether the accessing user is a super user or not (line 11) . Note that although M3 implements token-based authorization, it still checks the accessing user together with input token.

In many cases, permission checks are done ad hocly by directly testing user variables in conditional statements, instead of invoking its wrapping APIs. Among all 95 MPC vulnerabilities, 32.6% of them (31) are fixed with ad hoc permission checks. For instance, in Figure 3, the vulnerability (YARN-8455) is fixed by directly testing the accessing user (lines 127 - 129). Note that it is unclear how to manually specify those ad hoc permission checks.

Finding 1: There are four different mechanisms which invoke permission checking APIs in different manners. All APIs implement permission checks by simply testing user variables in conditional statements. In many cases, privileged operations are guarded by this simple conditional test directly as ad hoc permission checks: 31 out of 95 studied MPC vulnerabilities are fixed by introducing ad hoc permission checks.

2.3.2 Privileged Operations. Among all 95 MPC vulnerabilities, 84 of them access user owned data (i.e., user-related variables) without permission checks. For instance, the two vulnerabilities in Figure 1 and Figure 3 manifest themselves with unguarded accesses to user owned data `blockPoolId` and `app`, respectively. User owned data can be directly created by a user, or can be transitively derived from other user owned data. Distributed systems often divide large data chunks (e.g., data created by a user) into small pieces. Those small

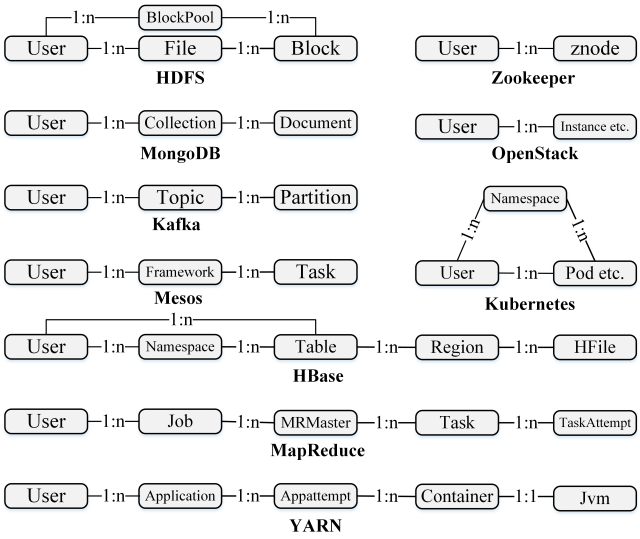

Figure 4: Simplified hierarchical structures of user-related types for all 10 studied systems. The structures are automatically inferred via a combined log-static analysis. For clarity, many user-related types are not given.

data pieces are managed by the system and typically invisible to regular users, i.e., they can only be accessed by the administrator for system maintenance. As an example, in HBase, a `Table` object is created by a regular user and can be divided into small regions (object of type `Region`). Both `Table` and `Region` are user-related types which are managed by the regular user and system user, respectively. Accesses to variables of both types (`Region` and `Table`) need to be checked.

Figure 4 shows the hierarchical structure of user-related types for the 10 studied systems. Each square represents a user-related type, and edges represent relations between data of those types. For instance, in HBase, `Namespace` objects are directly created by a user, while `Table` objects can be directly created by a user, or derived from `Namespace` objects.

We find that for 80/84 (95.2%) vulnerabilities, the bug-triggering variables with unguarded accesses are printed in logs. As an illustration, in Figure 1, line 3 logs the user owned data `blockPoolId` together with the accessing user `remoteUser`. Another interesting finding is that more than half unguarded variables (50/84) are not directly created by users. Those derived variables can be easily ignored by developers and often not guarded properly, resulting in MPC vulnerabilities.

The rest 11 vulnerabilities modify critical system states without checking permissions. As illustrated in Figure 5, the instance field `shutdownInProgress` denotes the state whether the node is shutting down or not. If yes, all operations on this node (e.g., writing data) will be blocked. This variable can only be modified by the administrator. However, the public API `shutdownDatanode()` forgets to check the accessing user, leading to a MPC vulnerability. Note that the field `shutdownInPogress` is tested as an error condition

```
1  void shutdownDatanode(String remoteUser) {//RPC method
2      + checkSuperuserPrivilege(remoteUser); // permission
       ↪  check
3      // Shutdown  can only be called  once.
4      if (this.shutdownInProgress) {
5        throw new IOException("Shutdown already in
         ↪  progress.");
6      }
7      this.shutdownInProgress = true; //privileged
       ↪  operation
8      ......
9  }
```

Figure 5: A vulnerability modifiying system-related variables (HDFS-CVE-2014-0229).

```
1  /*WARNING: the below methods can damage the cluster.*/
2  public void closeRegion(CloseRegionRequest request){
3      + getCoprocessorHost().preClose(request.getUser());
       ↪  //Permission check
4      String name = request.getRegion();
5      HRegion region = thisrsServices.getRegion(name);
6      CloseRegionHandler crh = new
       ↪  CloseMetaHandler(region);
7      this.executorService.submit(crh);
8  }
9  //CloseRegionHandler extends EventHandler
10 public void process() throws IOException {
11     LOG.info("Close " + region);
12     this.onlineRegions.remove(region);
13 }
14 //permission check
15 public void preClose(String user) throws IOException {
16     ....
17     if (!user.equals(this.owner)) {
18         throw new AccessControlException();
19     }
20     ....
21 }
```

privileged operation

Figure 6: A MPC vulnerability invoving multiple components (HBASE-7331).

which leads to exceptions if test fails (lines 4 - 6). In fact, 10/11 unprotected system-related variables are tested as error conditions.

Finding 2: Among the 95 MPC vulnerabilities, 84 of them have unauthorized accesses to user-related variables and 80/84 of these user-related variables are printed in logs. The rest 11 vulnerabilities modify critical system-related variables, where 10 system-related variables are used as error conditionss.

2.3.3 Component Interactions. There are 28 vulnerabilities directly involving multiple components, which means either the privileged operation or its fixing permission checks are performed remotely on a different component. Figure 6 gives an example (the bug and that in Figure 1 are grouped together and assigned one CVE). The API `closeRegion()` is a RPC function which attempts to offline the region and is supposed to be accessible to the administrator

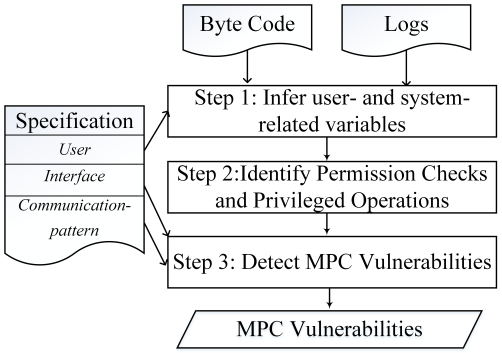

Figure 7: Overview of MPCHECKER.

```
1   Specification
2   User:
3      user1:hadoop,hbase:hadoop,yarn:hadoop
4   Interface: BlockingInterface
5   RPC:
6      Client:BlockingStub
7      Server:MasterRpcServices
8   Event:
9      context:null
10     Client:ExecutorServer.submit($1)
11     Server:$1.process()
```

Figure 8: Simplified specification example for HBase.

only. The missing permission check here indicates that any user can remove arbitrary region in HBase, causing huge damage to the system. Interestingly, even the developers have commented that this method can damage the cluster, they still forgot to check its permissions.

In this example, the privileged operation is performed remotely. The API `closeRegion()` wraps the request as a `CloseMetaHandler` event (line 6) and submits it at line 7. The event is then dispatched to its corresponding handler, i.e., `CloseRangionHandler.process()`, where the privileged operation at line 12 is invoked to remove regions from the system. Hence, the event-sending call at line 7 is eventually handled by its corresponding event handler. To preserve such semantics, we need to link the event-sending call to its corresponding handler as in [43].

Note that although only 28/95 vulnerabilities directly involve multiple components, interactions between component are prevalent in all studied systems. Some of those interactions may contain permission checks or privileged operations. There will be false positives and false negatives if those remote permission checks and privileged operations are not taken into account. Thus, it is critical to precisely analyze inter-component communications for detecting MPC vulnerabilities in distributed systems.

***Finding 3*:** 28/95 (29.5%) vulnerabilities directly involve multiple components.

3 MPCHECKER

Figure 7 overviews MPCHECKER at a high level. The tool consists of 3 steps. First, we automatically infer user- and system-related variables by analyzing run time logs of the studied system, together with its byte-code (currently the tool can only analyzes Java programs). Next, permission checks and privileged operations are identified from the set of inferred variables. In the last step, those privileged operations which are not guarded by permission checks are reported as MPC vulnerabilities if they are reachable from public interfaces .

MPCHECKER requires manual specification of user names, public interfaces, and inter-component communication patterns. Figure 8 gives a simplified specification for HBase. User names are needed to infer user-related variables. The given interfaces (i.e., `BlockingInterface`) define entry points of the system which can be remotely invoked by users. As in [45, 65], public methods implementing specified interface methods are APIs remotely accessible to users. Communication patterns (RPC, message events, and web sockets) are specified as a client/server pair, denoting the request-sending client and the request handler, respectively. We specify RPC by providing a client class with its corresponding server class (lines 5 - 7), since RPCs are realized via a client class and a corresponding server class implementing the same interface. Message events and web sockets are defined by providing a context-sensitive client site paired with its handler. The context denotes calling contexts such as containing functions and case conditions. A `null` context suggests that the handler is resolved to the same target under any calling context. We use a meta variable $1 to symbolize the actual parameter type of a client request, which is used to resolve the corresponding handler of a client (lines 8 - 11).

Let us examine the example in Figure 6. The call to method `this.executorService.submit(crh)` at line 7 is an event client. Since the argument `crh` has type `CloseRegionHandler`, the event handler is resolved as `CloseRegionHandler.process()`.

3.1 Infer User- and System-related Variables

As pointed out in Finding 2, most user-related variables are printed in logs and system-related variables are often tested as error conditions. Hence, we examine run time logs, together with a type-based static analysis, to discover user-related variables. Instance fields checked as error conditions are regarded as system-related variables. Alternatively, we could employ a pure static approach by statically examining every logging statement. However, it will require users to manually specify all user-representing variables, which is labor-intensive and error-prone.

3.1.1 User-related Variables. To start with, we need to figure out which variable represents a user. This is realized by a log analysis [63]. The log analysis constructs a regular expression pattern for each logging statement where string literals are kept in the expression and logged variables are abstracted as "`.*`". This pattern is then matched with run-time logs to extract the concrete run-time values of logged variables from matching log instances. Variables which have identical values with given user names are user-representing variables.

We illustrate how our log analysis works using the simple example (extracted from HBase) in Figure 9 and Figure 10, where Figure 9 shows 3 log statements and their run-time log instances

```
1  LOG.info(user+ "created table " + table);
2  //".*" created table ".*"
3  LOG.info("created region " + region);
4  //created region ".*"
5  LOG.info("Initialized assign procedures:table=" + table1
   ↪  + ",region=" + region);
6  //Initialized assign procedures:table= ".*" ,region= +
   ↪  ".*"
```

Figure 9: Simplified log statements and their logging patterns (in comments) for HBase.

```
1  user1 created table 'table1'
2  user1 created table 'table2'
3  created region "region1"
4  created region "region2"
5  Initialized assign
   ↪  procedures:table='table1',region='region1'
6  Initialized assign
   ↪  procedures:table='table1',region='region2'
```

Figure 10: Simplified run time log instances for HBase.

are given in Figure 10. Our studied systems use common logging libraries such as SLF4J[3] and Log4j[2], with provided logging interfaces including `fatal, error, warn, info, debug,` and `trace`. For each log statement, we extract a log pattern expressed by regular expressions. In Figure 9, the extracted log pattern for each statement is highlighted in comments. Variables printed in logs are abstracted as `".*"` in the log pattern, and their run-time values can be obtained from a matching log instance (as in [63]). For instance, the log pattern ⟨`".*" created table ".*"`⟩ matches with the log instance ⟨`"user1 created table 'table1'"`⟩. As a result, the run-time values of logged variables `user` and `table` are extracted as `"user1"` and `"'table1'"`, respectively. The variable `user` represents a user with name `"user1"`.

Rule 1. *A variable is a user-related variable if it is printed in a same log instance with a user variable or with another user-related variable.*

The above rule is applied to discover user-related variables from logs. For audit or diagnosis purpose, systems often log events when a user accesses his data, or when user data is further partitioned or moved. Hence, it is natural to regard those variables as user-related variables if they appear in a same log instance with user variables. Such variables often represent data directly created by users, which are accessible to their owner only. Similarly, a variable in a same log instance with another user-related variable suggests that it is derived from the other user-related variable.

In our example in Figure 10, variable `table` and `user` appear in the same log instance (lines 1 and 2). Hence, `table` is a user-related variable. Note that the variable `table` have multiple run time values (i.e., `"'table1'"` and `"'table2'"`), suggesting that a user can create multiple tables at run time. The variable `region` is printed in same log instances with `table` (line 5 and 6), and is considered as a user-related variable derived from `table`.

Rule 2. *A variable is a user-related variable if its type matches with the type of a user-related variable. Type T_A matches with T_B if 1) T_A is an object type identical to or inherits from T_B, 2) if it is a collection with one element type matches with T_B, or 3) if it has an indexing field whose type matches with T_B.*

Our log analysis effectively discovers user-related variables from log instances. However, some user-related variables may not be logged or their logging statements may not have been executed. Hence, we perform a simple type-based analysis, as summarized in the above rule, to find all other user-related variables in a program. The type matching rules for the first two cases are self-explainable. The third case is to handle the common practice when a field is used to uniquely index its containing object. For instance, in HBase, field `region` is used as an index for `RegionInfo`. As a result, `RegionInfo` is also regarded as a user-related type.

The type-based analysis applies to user-related variables with object types only. We employ the simple type-based analysis instead of a classic pointer analysis because user-related variables are often well typed, and precise and sound pointer analysis [41, 42] for distributed systems remains to be an open research topic. For variables of primitive types, we following their use-def chains to obtain the set of variables they propagate to. Specifically, we examine the def-use chain of a primitive-typed variable, to discover those variables/fields it is assigned/stored to. This process is iteratively applied to those newly discovered variables until a fixed point. As such, all variables (fields) transitively assigned from a user-related variable are discovered.

3.1.2 System-related variables. We find system-related variables based on the observation that they are often tested as error conditions. Hence, we first locate those error-indication program points which abort normal execution.

```
1  if (this.stopped) {
2     LOG.error("Node has been stopped");
3     return;
4  }
```

We consider the following two types of error-indication program points: 1) statements throwing exceptions (e.g., line 5 in Figure 5), and 2) return statements of a public API immediately following a faulty-level logging statement (e.g. `Log.error()`), as in the above code snippet.

Given an error-indication point, we examine the conditional statement it directly control depends on. If the conditional statement checks an instance field, we regard the field as a system-related variable. The intuition is that system states are commonly stored in instance fields. Conditional statements may also test input conditions (e.g., parameters or function returns), and these checked variables are not counted as system-related variables.

3.2 Identify Permission Checks and Privileged Operations

3.2.1 Permission checks. As stated in Finding 1, permission checks are implemented by either directly testing user variables or calling APIs wrapping these user tests. Hence, all conditional statements testing user variables are permission checks. Permission checking

APIs are recognized via function summary: a method is a permission checking API if there exists a permission check post-dominating its entry. For instance, `checkSuperuserPrivilege()` in Figure 1 and `preClose()` in Figure 6 are summarized as permission checking APIs.

Sometimes, an API performs permission checks only if the authorization service is enabled, as shown in the code snippet below.

```
1  void checkSuperuserPrivilege(String user){
2    RouterPermissionChecker pc = getPermissionChecker();
3    if (pc != null) {
4      pc.checkSuperuserPrivilege(user);
5    }
6  }
```

When security configuration is not enabled, the authorization object `pc` is null and the check at line 4 is skipped. Without considering this case, we will report many MPC vulnerabilities which can be triggered only if the security configuration is disabled. Developers believe such configuration is intentional and often classify them as false positives. To address this case, we adopt a similar strategy as in [65]: when a permission checking invocation is guarded by a null check on its receiving object, we mark the null check (i.e., line 3 in the example) as the wrapper permission check.

3.2.2 Privileged Operations. Accesses to user- or system-related variables are categorized as privileged operations. More specifically, those operations writing user- or system-related variables, or leaking user-related variables are privileged operations. Intuitively, they suggest dangerous operations that may corrupt user data or the whole system, or may leak sensitive information.

Write operations are `putField` instructions (in Java byte-code) on user- or system-related variables. If the variable is a collection, invocation to APIs modifying the collection (by removing or adding elements, e.g., `insert` and `remove` for `set`) are manually labeled as write operations. User-related variables may contain sensitive information private to the owner. Hence, a statement is a privileged operation if it returns a user-related variable from a public API, or if it writes user-related variable to IO. Since such operation may disclose sensitive information to public, it is labeled as a leak operation. We optimistically assume that variables with `int` or `bool` types contain no sensitive data, and filter out those leak points for `int`- or `bool`-typed variables.

3.3 Detect MPC Vulnerabilities

A MPC vulnerability manifests if there exists a program path from the program entry (entry of public APIs) to a privileged operation. We formulate the detection of MPC vulnerabilities as a classic IFDS (inter-procedural, finite, distribute, subset) dataflow problem [51], which effectively computes whether a privileged operation is guarded by a permission check or not. An unguarded privileged operation suggests that there exists a path from the entry to the privileged operation without permission checks.

3.3.1 Inter-component Communications. One of the key challenges in analyzing distributes systems lies in how to effectively handle inter-component communications [9]. Components communicate with each other via RPCs, message events, or web sockets. To avoid the daunting tasks of analyzing the complex communication mechanism implementations (where dynamic features such as reflection are heavily used), we rely on user specification (Figure 8) to resolve the target of a client request. A request can then be viewed as a direct call to its handler. As such, different components of a system can be connected together and the system can be analyzed in the standard manner.

However, it is often impractical to analyze a distributed system as a whole, given the large combined program size of all its components. Hence, instead of linking all components together, we apply function summary to summarize whether a request is handled by a permission checking API or by a privileged operation. A request is regarded as a privileged operation if there are unguarded privileged operations in its corresponding handler. Otherwise, it is a permission check if there exists a permission check which post-dominates the entry of the handler. Take Figure 6 for example. The event-sending client at line 7 (`this.executorService.submit()`) is summarized as a privileged operation since there exists unguarded privileged operation (line 12) in its handler (`CloseRegionHandler.process()`). With this optimization, we can analyze each component separately, significantly reducing program size and improving efficiency.

3.3.2 Data-flow Analysis. The IFDS (inter-procedural, finite, distribute, subset) algorithm by Reps et al. [51] solves a large set of inter-procedural data-flow problems as a generalized graph reachability problem. The algorithm operates on a so-called exploded graph. At each program point in the inter-procedural call graph (ICFG) of a program, a node is introduced to the exploded graph for each data-flow fact in the analysis domain. Edges encapsulate the semantics of transferring functions in data-flow analysis. A data-flow fact holds at a program point if there exists a realizable path from the program entry to its representing node in the exploded graph. For context-sensitivity, return from a function needs to match with its call-site to make the path realizable.

For MPC vulnerability detection, the data-flow value is either *true* or *false*, denoting whether a point is unguarded or not, i.e., whether there exists a path from program entry to the point without checking permissions or not. Equations 1-3 give the data flow functions. The analysis propagates data-flow values forwardly along the ICFG of the program. At the entry of a public interface, the value is initialized to *true* and it is only set to *false* by permission checks. At a program point when multiple paths converge, data-flow values along different paths are merged with the $\vee$ operator. If the analyzed result is *true* at a privilege operation, a MPC vulnerability will be reported.

$$IN_{entry} = true \tag{1}$$

$$IN_i = \bigvee_{p \in pred_i} (OUT_p) \tag{2}$$

$$OUT_i = \begin{cases} false & \text{permission check} \\ IN_i & \text{otherwise} \end{cases} \tag{3}$$

3.3.3 An example. Let us examine the example in Figure 11. The API `move` is a public API which can be remotely invoked to move the given region to a target node. If the target node is given, a

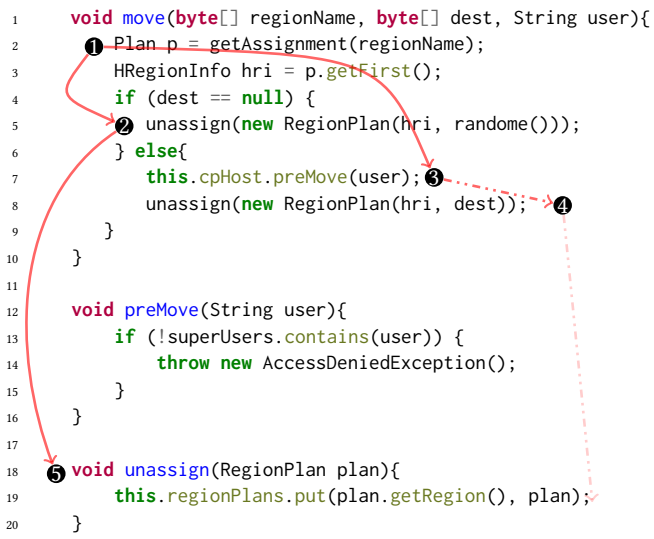

Figure 11: An illustration example (HBASE-6246).

Table 2: Number of user specifications. # Is is the number of specified interfaces. # T is the total number of specified communication patterns.

System	# Is	# APIs	# Communication patterns			
			# RPCs	# Events	# Sockets	# T
HDFS	3	732	31	2	6	39
YARN	2	542	21	115	52	188
MapReduce	2	155	15	54	17	86
HBase	2	371	27	17	30	74
Zookeeper	1	7	1	19	43	63
CloudStack	2	1994	134	0	11	145

permission check is applied at line 7 by calling the API `preMove()`. However, if the target node is not given, the API randomly chooses a node, and the region is moved without checking permissions.

In our analysis, at program point ❶, the data-flow fact is initialized as *true* then being propagated forwardly. At the call site ❷, IFDS propagate the fact to the callee function `unassign()`, and the program point ❺ at line 19 is an unguarded privileged operation on this path. On another path, at program point ❸ (line 7), the fact is set to *false*. Thus, on the path from ❸ → ❹ → ❺, the fact remains *false*. At the merge point ❺, since the data-flow fact is *true* along the path ❶ → ❷ → ❺, the fact at line 19 is *true* after merging facts along different paths. Hence, we successfully report a vulnerability.

3.4 Limitation

Manual Efforts. One of the concerns is the manual efforts involved in specifying public interfaces and communication patterns. Table 2 presents the number of specification for the 6 Java systems we evaluated. Column 1 gives the number of specified public interfaces, which is at most 3 (HDFS). Public APIs implementing those interface methods are automatically derived, and their numbers are much larger (Column 3). Compared to interfaces, it takes more efforts to specify communication patterns: we manually provide 188 communication patterns for YARN and 145 for CloudStack (Column 7). YARN dispatches events and web sockets to different handlers at distinct contexts. As a result, an event or socket client is often cloned multiple times, each with a distinct context. On the other hand, CloudStack mostly communicate via RPCs and the system offers a large number of public services via RPC classes (134). In our experience, developers can easily give a specification of communication patterns of a system: it takes us at most one day to specify the communication patterns for a system, by reading its official documents.

Permission Checks. We only consider the simple permission checking implementation where a user is conditionally tested. This trade-off works well for the 10 distributed systems under study. However, it may not be generalizable to other implementations, e.g., token-based authorization with permission checked against a token. Unrecognized permission checks can be manually specified, as in previous work. In our study, systems adopting token-based authorization check both user and its token together.

Our approach optimistically assumes that any permission check is sufficient to prevent unauthorized accesses. It does not apply to authorization vulnerabilities caused by insufficient permission checks or misused permission checks [28]. We plan to extend our approach to detect such vulnerabilities, by analyzing the identity and capability of checked users.

Privileged Operations. We introduce a novel log-based analysis to automatically identify privileged operations from inferred user- and system-related variables, based on the assumption that accesses to those variables are privileged. One may argue that this assumption is not true: system-related variables may be harmlessly modified, and user-related variables can be exposed to public without disclosing confidential information. There is no simple yes-or-no answer to the argument. However, in our experiments, only 5 false positives are caused by incorrect privileged operations, suggesting the truthfulness of our assumption.

In our evaluation, the log-based analysis does discover significantly more privileged operations than existing consistency-checking analyses. However, its effectiveness largely depends on the quality of logs. Distributed systems tend to offer rich log information for diagnosis [66], and we are able to precisely uncover hundreds of privileged operations from a small number of log instances generated by built-in workloads.

4 EVALUATION

We implement MPChecker in WALA[6], with 9K lines of Java code. The standard insensitive Andersen's analysis is employed to build the call graph and we extend WALA's IFDS framework for MPC vulnerability detection (Step 3 in Figure 7). We evaluate MPChecker using 6 distributed systems written in Java (Table 2), including HDFS, HBase, MapReduce, YARN, Zookeeper, and CloudStack [36]. Note that CloudStack is not a subject in our empirical study and this bemchmark is used in our evaluation to test the generality of our approach. All experiments are conducted on a server with an 80-core Intel(R) Xeon(R) Gold 6230 CPU at 2.10GHz and 500 GB of memory, running CentOS 8.

Table 3: Number of privileged operations and permission checks. # U and # S stands for user-related and system-related, respectively. # PC is the number of permission checks.

System	Related Variable			Privileged Operation			# PC
	# U	# S	# Total	# U	# S	# Total	
HDFS	417	270	687	564	217	781	18
YARN	1,045	59	1,104	611	331	942	17
MapReduce	1,286	210	1,496	317	157	474	21
HBase	659	151	810	511	139	650	14
Zookeeper	98	53	151	82	27	109	2
CloudStack	3,426	2,084	5,510	664	227	891	130

For each system under evaluation, we pull its latest trunk version to evaluate the capability of MPChecker in detecting new MPC vulnerabilities. Each system is profiled with its built-in workload to collect run-time logs. We evaluate the recall of MPChecker by applying the tool on previous buggy versions with known vulnerabilities. Since all previous work [19, 54, 56, 65] targets different applications and is not publicly available, there is no available baseline to compare. We try our best to compare with previous approaches by manually verifying whether a reported bug can be detected with previous consistency-checking mechanisms or not.

The evaluation answers the following research questions:

- RQ1. How precise does MPChecker identify privileged operations?
- RQ2. How effective does MPChecker detecting MPC bugs?
- RQ3. How efficient is MPChecker?

4.1 RQ1. Identify Privileged Operations

Whether we can precisely identify privileged operations or not is key to the effectiveness of our tool. Table 3 summarizes the number of related variables (Columns 2 - 4), the number of privileged operations (Columns 5 - 7), and the number of derived permission checking APIs (Column 8) for each system. MPChecker automatically infers thousands of user-related (Column 2) and system-related (Column 3) varaibles. Accesses to those variables generate hundreds of privileged operations as shown in Column 7.

An immediate question arises: are those identified operations really privileged? We have manually checked each operation and confirmed that all of them do access user- or system-related variables as expected. Later experiments further confirmed that most of those accesses are real privileged operations: there are only 5 false positives introduced by incorrect privileged operations.

It is difficult to measure whether there is any missed privileged operation. We manually examined the bug-triggering operations in all studied vulnerabilities belonging to the systems under evaluation: one privileged operation is missed because it accesses a user-related variable not printed in logs.

4.2 Effectiveness

We evaluate the effectiveness of MPChecker in terms of its ability in detecting existing and new vulnerabilites.

4.2.1 Existing Vulnerabilities. Table 4 summarizes the results in detecting existing vulnerabilities. All 57 MPC vulnerabilities from the 5 Java systems in our study are selected. These 57 vulnerabilities

Table 4: Results on 57 existing vulnerabilities. Number in bracket denotes the number of bugs in a issue.

Detected	HDFS-3628 MR-7097 HDFS-15053(2) HDFS-3331 HDFS-2917 CVE-2014-0229(3) YARN-7157 YARN-3517 YARN-8319(2) YARN-8221 YARN-5554 YARN-8455(6) HBASE-6246 HBASE-12552 HBASE-7331(4) HBASE-8692 HBASE-12674 HBASE-19400(6) HBASE-24345 HBASE-19634 HBASE-19401 HBASE-19483(9) HBASE-15132 HBASE-6292 ZOOKEEPER-CVE-2019-0201
Not Detected	HBASE-15488 HBASE-12916 HBASE-19400(4) HBASE-6104

access 49 different variables, 21 different user/resource types and 54 different operations. Except for the user-related variable `WALEntry`, which is never logged, our analysis successfully identifies all user-related variables and their associated types and operations. Overall, MPChecker reports 50 existing vulnerabilities with 7 false negatives, at a recall rate of 87.8%.

Four vulnerabilities (HBASE-6104 and 3 in HBASE-19400) are not detected due to reflection, where MPChecker fails to find the privileged operations in reflective method calls. Static analysis for reflection [37] is a well-known challenging problem. The vulnerability HBASE-12916 is missed because it accesses user-related variables which are not printed in logs. This is also the only known case where our log-based analysis fails to uncover user-related variables. For the two remaining undetected vulnerabilities (HBASE-15488 and 1 bug in HBASE-19400), privileged operations are conducted on a different system (Zookeeper). Currently MPChecker does not take such inter-system interactions into account.

4.2.2 New Vulnerabilities. Table 5 gives the 44 new vulnerabilities reported by MPChecker. For each reported vulnerability, we list the public API exposing the bug, type of privileged operation (Column 3), related variable (Columns 4 - 6), and number of involved components (Column 7).

Except for Zookeeper, MPChecker detects MPC vulnerabilities in all systems under evaluation: HDFS has by far the most number of vulnerabilities (24), followed by HBase (8) and CloudStack (8). Most detected vulnerabilities (36) are caused by unprotected privileged write operations, and the rest 8 vulnerabilities leak user owned data (Column 3). Column 5 lists the related variable for each vulnerability: 16/28 related variables are primitive/objective typed (Column 4), and 27/17 variables are user-related/system-related (Column6). As shown in Column 7, 20 bugs involve at least 2 components and HDFS-15752-1 is triggered by 3 components together.

We have patched all 44 reported vulnerabilities. Among the 44 bugs, 20 of our patches have been accepted and another 23 bugs have been confirmed by the original developers (Column 8). The bug MapReduce-7330-43 is still under review. All confirmed bugs are labeled as *critical* bugs and can be easily exploited.

A PoC Exploit. Figure 12 constructs a proof-of-concept (POC) exploit for HBASE-25432-25, with 4 lines of code. The exploit works as follows. First, we connect to the server as *user2* (line 1). At line 2, we forge a request to disable the table named "`test`". Note that

Table 5: New vulnerabilities reported by MPChecker. Each vulnerability is indexed with "issue ID-number", where multiple vulnerabilities are grouped in one issue. The issue numbers for HBASE-27 is not disclosed, as requested by the original developers. # Comp stands for the number of involved components.

ID	API	Operation	Related Types	Related Variables	User/System	# Comp	Status
HDFS-15752-1	fsck	Leak	BlockId	blocksMap	user-related	3	Fixed
HDFS-16004-2	isFormatted	Write	Journal	journalsById	system-related	1	Confirmed
HDFS-16004-3	getJournalState	Write	Journal	journalsById	user-related	1	Confirmed
HDFS-16004-4	newEpoch	Write	PersistentLongFile	lastPromisedEpoch	system-related	1	Confirmed
HDFS-16004-5	format	Write	StorageState	state	system-related	1	Confirmed
HDFS-16004-6	journal	Write	long	curSegmentTxId	user-related	1	Confirmed
HDFS-16004-7	heartbeat	Write	StorageState	Value	system-related	1	Confirmed
HDFS-16004-8	startLogSegment	Write	PersistentLongFile	lastWriterEpoch	user-related	2	Confirmed
HDFS-16004-9	finalizeLogSegment	Write	long	nextTxId	user-related	2	Confirmed
HDFS-16004-10	purgeLogsOlderThan	Write	Journal	journalsById	user-related	2	Confirmed
HDFS-16004-11	getEditLogManifest	Leak	RemoteEditLog	fjm	user-related	1	Confirmed
HDFS-16004-12	prepareRecovery	Leak	PersistentLongFile	lastWriterEpoch	user-related	1	Confirmed
HDFS-16004-13	acceptRecovery	Write	long	highestWrittenTxId	user-related	2	Confirmed
HDFS-16004-14	doPreUpgrade	Write	BestEffortLongFile	committedTxnId	user-related	2	Confirmed
HDFS-16004-15	doUpgrade	Write	long	cTime	user-related	2	Confirmed
HDFS-16004-16	canRollBack	Write	Journal	journalsById	system-related	1	Confirmed
HDFS-16004-17	doRollback	Write	StorageState	state	system-related	1	Confirmed
HDFS-16004-18	discardSegments	Write	BestEffortLongFile	committedTxnId	user-related	1	Confirmed
HDFS-16004-19	getJournalCTime	Write	Journal	journalsById	user-related	1	Confirmed
HDFS-16004-20	getEditLogManifestFromJournal	Leak	RemoteEditLog	fjm	user-related	1	Confirmed
HDFS-16004-21	getJournaledEdits	Leak	NavigableMap	dataMap	user-related	1	Confirmed
HDFS-16004-22	BackupNode#startLogSegment	Write	BNState	bnState	user-related	2	Confirmed
HDFS-16004-23	BackupNode#journal	Write	long	txid	user-related	1	Confirmed
HDFS-16004-24	doFinalize	Write	Journal	journalsById	user-related	1	Confirmed
HBASE-25432-25	setTableStateInMeta	Write	TableName	tableName2State	user-related	1	Fixed
HBASE-25432-26	fixMeta	Write	RegionStateNode	regions	user-related	1	Fixed
HBASE-27	-	Write	Region	-	user-related	1	Fixed
HBASE-25441-28	stopServer	Write	boolean	stopped	system-related	1	Fixed
HBASE-25441-29	updateFavoredNodes	Write	Address	regionFavoredNodesMap	user-related	1	Fixed
HBASE-25441-30	updateConfiguration	Write	Configuration	conf	user-related	1	Fixed
HBASE-25441-31	clearRegionBlockCache	Leak	Region	onlineRegions	user-related	2	Fixed
HBASE-25441-32	clearSlowLogsResponses	Leak	State	states	user-related	2	Fixed
HBASE-25877-33	compactionSwitch	Write	boolean	compactionsEnabled	system-related	2	Fixed
CLOUDSTACK-10434-34	updateVolume	Write	boolean	_txn	system-related	2	Fixed
CLOUDSTACK-10434-35	detachVolumeViaDestroyVM	Write	boolean	_txn	system-related	2	Fixed
CLOUDSTACK-10434-36	takeSnapshot	Write	boolean	_txn	system-related	2	Fixed
CLOUDSTACK-10434-37	migrateVolume	Write	boolean	_txn	system-related	2	Fixed
CLOUDSTACK-10434-38	createApiKeyAndSecretKey	Write	boolean	_txn	system-related	2	Fixed
CLOUDSTACK-10434-39	applyLBStickinessPolicy	Write	boolean	_txn	system-related	2	Fixed
CLOUDSTACK-10434-40	applyLBHealthCheckPolicy	Write	boolean	_txn	system-related	2	Fixed
CLOUDSTACK-10434-41	createPrivateTemplate	Write	boolean	_txn	system-related	2	Fixed
CLOUDSTACK-10434-42	updateSnapshotPolicy	Write	boolean	_txn	system-related	2	Fixed
MAPREDUCE-7330-43	getJobAttempts	Leak	ApplicationAttempt	amInfos	user-related	1	Submitted
YARN-10555-44	getAppAttempts	Leak	ApplicationAttempt	applications	user-related	1	Fixed

```
1 Hbck hbck =/*login as user2*/
  ↪ ConnectionFactory.createConnection().getHbck();
2 TableState state=new/*'test' belong to user1*/
  ↪ TableState(TableName.valueOf("test"),
  ↪ TableState.DISABLED);
3 hbck.setTableStateInMeta(state);
4 hbck.tableExists(TableName.valueOf("test"));
```

Figure 12: The PoC example.

this table belongs to another user (*user1*), which is not supposed to be accessed by *user2*. After receiving the request (sent at line 3), the server will disable the table as requested without checking its permissions. Line 4 confirms that the table named "test" has been dropped. As such, an attacker can easily corrupt data belong to other users, resulting in severe damages.

Security Impacts. MPC vulnerabilities can cause security problems such as denial of service (DoS), data corruption, and information leak (see Section 1.3). Among the 44 reported bugs, 7 of them cause DoS, 9 of them leak sensitive information (e.g., MapReduce-7330-43 leaks password in amInfos), and the rest 28 bugs cause data loss. All those bugs can be exploited easily and are confirmed as security flaws by the original developers. Particularly, the developers requested to undisclose 1 bug since they believe that the bug is a severe flaw which may cause catastrophic problems.

Fixes. All 44 new vulnerabilities are fixed by introducing permission checks, and 23 of them are patched with ad hoc permission

Table 6: Analysis time of MPChecker.

System	(#) LOC	(#) logs	(s) Step1	(s) Step2	(s) Step3	(h) Total
HDFS	870,746	809	56	34	20,769	5.79
YARN	946,766	792	79	81	23,400	6.54
MapReduce	32,2968	1,089	103	48	4,871	1.39
HBase	1,257,954	662	68	79	26,351	7.32
Zookeeper	363,853	216	13	8	120	0.04
CloudStack	671,842	999	1,693	842	59,216	17.15

checks since there is no existing API available. Once again, this emphasizes that it is difficult to manually specify all permission checks in distributed systems. Our patches also help to find two new bugs in the security test of HBase: the newly introduced permission checks throw access denied exceptions which are not correctly handled. The two bugs have already been fixed.

Comparison. We manually verify each reported vulnerability and confirm that only 13 of them can be detected with existing consistency-checking mechanisms [21, 56, 65]. These mechanisms cannot identify privileged operations for the remaining 31 vulnerabilities: 26 of which are not guarded by any permission checks, 4 privileged operations are remotely performed on a separate component, and 1 operation is partially guarded by an ad hoc permission check. In addition, if ad hoc permission checks are not recognized, an analysis will report hundreds of false positives.

4.2.3 False Positives. MPChecker reports 7 false positives, with a false positive rate of 13.7%. There are 5 false positives in HBase. Two of them are caused by permission checks performed in a different system (i.e., HDFS): the two APIs `bulkLoadHFile` and `cleanupBulkLoad` access files in HDFS, where permissions are checked by HDFS. Such false positives can be addressed with additional specification, by marking those corresponding invocations to HDFS as permission checks. MPChecker reports another 2 false positives because they return user-related variable of type `Region`. Theoretically, an attacker can construct a DDOS attack once the target node of the region is located. However, developers believe such information should be made to public because *"locate regions by any client is fundamental to how HBase works"*. The last false positive in HBase returns cluster ID to user, which is considered to be safe. MPChecker reports it as a bug because it assumes such user-related variables may contain sensitive information.

The rest 2 false positives are reported in YARN: the two APIs (`getNewApplication` and `getNewReservation`) can increase the application ID in YARN. MPChecker reports them as bugs since they modify user-related variables. However, YARN is designed in such a way that any authenticated user can submit an application hence the permission always granted. The 2 reports in YARN, together with the last 3 false reports in HBase, are the only 5 cases caused by incorrect privileged operations with harmless accesses to user-related variables.

4.3 Efficiency

Table 6 reports the times in analyzing each system. Column 1 presents the number of lines of code (LOC) for each system and Column 2 gives the number of log instances generated by running the built-in workload of each system. The number of log instances are small: except for MapReduce (1,089), all other systems have less than 1,000 log instances. Column 3 - 5 show the break down times of each step and the total analysis times are given in Column 6. The majority of analysis times are consumed by the IFDS solver in step 3 (Column 5).

The analysis time is directly related to the number of public APIs of each system (Column 3 in Table 2), since we run the time-consuming IFDS solver once for each API. It takes the longest time to analyze CloudStack (17.15 hours), with 1,764 APIs. Zookeeper can be analyzed in 3 minutes because it has only 7 public APIs.

```
1  public void fsck(){
2    ......
3    for (String blk: this.blockIds) {
4      BlockInfo blockInfo =
       ↪  this.blockManager.getStoredBlock(block);
5      this.out.println(blockInfo);
6    }
7    .....
8  }
9  Output:
10   Block Id: blk_1073741825
11   Block belongs to: /private/file_name_sensitive.txt
```

Figure 13: A vulnerability leaking sensitive information (HDFS-15752-1).

4.4 Case Studies

Here we study 2 interesting new bugs reported by MPChecker.

HDFS-15752-1. MPChecker reports this bug in the web API interface `fsck` since it leaks user-related variable (`blockInfo`) at line 5. The API returns all block information (including information for blocks belonging to other users) to the request sender, including block ID and its containing file as shown in the output (lines 9 - 11). However, the file name and location may be confidential to the requesting user. As an illustration, the file may belong to another user and it is stored in a confidential directory (`private`) with access mod set to x700. By exploiting this vulnerability, an attacker can invoke the API `fsck` to read secret files names in the confidential directory, bypassing access control of the file system. We have submitted this issue to the Apache security group and they are considering to assign it a CVE number.

HBASE-25441-32. This bug is reported by MPChecker because the user-related variable `state` is returned from a public API (line 7). Hence, it may leak sensitive information. Further investigation of the buggy code tells us that the function call to `evictBlockByHfileName()` (line 5) is an expensive operation (commented in line 9). As such, an attacker can also invoke this API to perform expensive operations on the server node numerous times, causing denial of service.

4.5 Discussion

MPChecker effectively detects MPC vulnerabilities at high precision, with a recall and false positive rate of 87.8% and 13.7%, respectively. The effectiveness of MPChecker is largely benefited from its ability in precisely identifying privileged operations. Among the

```
1  public ClearRegionBlockCacheResponse
   ↪ clearRegionBlockCache(Request request) {
2    List<State> states = new ArrayList<State>();
3    List<HRegion> regions =
     ↪ getRegions(request.getRegionList(), stats);
4    for (HRegion region : regions) {
5      stats = stats.append(evictBlocksByHfileName(region));
6    }
7    return states;
8  }
9  /*Evicts all blocks for a specific HFile. This is an
   ↪ expensive operation... */
10 public CacheEvictionStats evictBlocksByHfileName(){
11     .....
12 }
```

Figure 14: A vulnerability causing DOS (HBase-25441-32).

44 reported new bugs, 31 of them are undetectable with existing techniques because their consistency-checking analyses fail to find many privileged operations which are never protected. In addition, only 1 false negative is observed due to missed privileged operations and only 5 false positives are caused by incorrect privileged operations, suggesting the precision of our log-based analysis in identifying privileged operations.

As other bug detectors, MPChecker does not guarantee soundness or completeness. Hence, there are false positives and false negatives. All false reports are due to well-known challenges such as reflection and interactions cross multiple systems.

5 RELATED WORK

There are several approaches [28, 53, 54, 56, 65] targeting MPC vulnerabilities in different systems. AutoISES [56] automatically infers code-level security specifications with user-provided security checks, by statically analyzing the correlation between data structure accesses and security checks. Sun et al. [54] constructs a sitemap for different roles in a web application, and MPC vulnerabilities are detected by checking whether browsing from unprivileged pages to privileged pages succeeds or not. RoleCast [53] observes a consistent software engineering pattern and develops a novel algorithm for discovering this pattern in web applications. The tool then applies role-specific consistency analysis to find missing security checks. PeX [65] applies a static consistency-checking analysis to find critical Linux kernel functions frequently guarded by permission checks, then searches for a vulnerable path to an unprotected critical kernel function. ACHyb [28] addresses the false positives of Pex via a combined static-dynamic analysis. Compared to existing work, MPChecker targets distributed systems. In MPChecker, privileged operations are automatically identified by inferring user- and system-related variables via a log-based analysis. This approach effectively addresses the limitation of existing consistency-checking approaches in analyzing distributed systems.

Bug detection for distributed cloud systems has been extensively studied in the past [8, 15, 24, 24, 25, 32, 38, 43, 44]. Those state-of-the-arts focus on crash recovery bugs [22, 44, 46], network partition bugs [7], distributed concurrency bugs [39, 43, 45], exception-related bugs [14, 64], and data corruption bugs [18]. To the best of our knowledge, MPChecker is the first tool targeting MPC vulnerabilities in distributed systems.

6 CONCLUSION

We present MPChecker, the first MPC vulnerability detection tool for distributed systems. MPChecker applies a log-based analysis to infer use- and system-related variables, whose accesses are privileged operations. We have evaluated MPChecker with 6 widely-used distributed systems. The tool reports 44 new critical vulnerabilities from 6 popular distributed systems, including 1 severe security flaws with details undisclosed to public.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable inputs. This work is supported by the National Science Foundation of China (NSFC) under grant number 62132020, and the CCF-Huawei Innovation Research Plan.

REFERENCES

[1] 2018. North Branford Man Who Hacked into More Than 200 Apple iCloud Accounts Sentenced to Prison. https://www.justice.gov/usao-ct/pr/north-branford-man-who-hacked-more-200-apple-icloud-accounts-sentenced-prison.
[2] 2019. Apache log4j, a logging library for Java. http://logging.apache.org/log4j/2.x/.
[3] 2019. Simple logging facade for Java (SLF4J). http://www.slf4j.org/.
[4] 2019. United States: Biggest Cloud Security Vulnerability Is Misuse Of Employee Credentials And Improper Access Controls! https://www.mondaq.com/unitedstates/security/822364/biggest-cloud-security-vulnerability-is-misuse-of-employee-credentials-and-improper-access-controls.
[5] 2020. Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/.
[6] 2021. wala/WALA: T.J. Watson Libraries for Analysis. https://github.com/wala/WALA.
[7] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An analysis of network-partitioning failures in cloud systems. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 51–68.
[8] Peter Alvaro, Joshua Rosen, and Joseph M Hellerstein. 2015. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 331–346.
[9] Henri E Bal, Jennifer G Steiner, and Andrew S Tanenbaum. 1989. Programming languages for distributed computing systems. *ACM Computing Surveys (CSUR)* 21, 3 (1989), 261–322.
[10] David Bernstein. 2014. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing* 1, 3 (2014), 81–84.
[11] Bharat Bhargava and Leszek Lilien. 2004. Vulnerabilities and threats in distributed systems. In *International Conference on Distributed Computing and Internet Technology*. Springer, 146–157.
[12] Dhruba Borthakur et al. 2008. HDFS architecture guide. *Hadoop Apache Project* 53, 1-13 (2008), 2.
[13] Thomas Brewster. 2017. Massive WWE Leak Exposes 3 Million Wrestling Fans' Addresses, Ethnicities And More. https://www.forbes.com/sites/thomasbrewster/2017/07/06/massive-wwe-leak-exposes-3-million-wrestling-fans-addresses-ethnicities-and-more/?sh=56477fb175dd.
[14] Haicheng Chen, Wensheng Dou, Yanyan Jiang, and Feng Qin. 2019. Understanding exception-related bugs in large-scale cloud systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 339–351.
[15] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2020. CoFI: consistency-guided fault injection for cloud systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 536–547.
[16] Kristina Chodorow. 2013. *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc.".
[17] Cyber. 2019. Information on the Capital One Cyber Incident. https://www.capitalone.com/digital/facts2019/.
[18] Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu, and Peipei Wang. 2018. Dscope: Detecting real-world data corruption hang bugs in cloud server systems. In *Proceedings of the ACM Symposium on Cloud Computing*. 313–325.
[19] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. 2009. Nemesis: Preventing Authentication & [and] Access Control Vulnerabilities in Web Applications. (2009).
[20] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[21] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. (Oct. 2001), 57–72.

[22] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An empirical study on crash recovery bugs in large-scale distributed systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 539–550.

[23] Bernd Grobauer, Tobias Walloschek, and Elmar Stocker. 2010. Understanding cloud computing vulnerabilities. *IEEE Security & privacy* 9, 2 (2010), 50–57.

[24] Haryadi S Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M Hellerstein, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. 2011. FATE and DESTINI: A framework for cloud recovery testing. In *Proceedings of NSDI'11: 8th USENIX Symposium on Networked Systems Design and Implementation*. 239.

[25] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. 2011. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 265–278.

[26] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. 2019. Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*. IEEE Press, 267–279. https://doi.org/10.1109/ASE.2019.00034

[27] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center.. In *NSDI*, Vol. 11. 22–22.

[28] Yang Hu, Wenxi Wang, Casen Hunger, Riley Wood, Sarfraz Khurshid, and Mohit Tiwari. 2021. ACHyb: a hybrid analysis approach to detect kernel access control vulnerabilities. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 316–327.

[29] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8.

[30] Amir Jerbi. 2019. Docker Hub Unauthorized Access Incident: What You Should Know. https://blog.aquasec.com/docker-hub-incident-container-encryption.

[31] O'Ryan Johnson. 2019. ConnectWise Hit In EU Ransomware Attack. https://www.crn.com/news/channel-programs/connectwise-hit-in-eu-ransomware-attack.

[32] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. 2007. Life, death, and the critical transition: Finding liveness bugs in systems code. NSDI.

[33] Steve Kosten. 2020. 7 Cloud Computing Security Vulnerabilities and What to Do About Them. https://towardsdatascience.com/7-cloud-computing-security-vulnerabilities-and-what-to-do-about-them-e061bbe0faee.

[34] Brian Krebs. 2018. USPS Site Exposed Data on 60 Million Users. https://krebsonsecurity.com/2018/11/usps-site-exposed-data-on-60-million-users/.

[35] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. 1–7.

[36] Rakesh Kumar, Kanishk Jain, Hitesh Maharwal, Neha Jain, and Anjali Dadhich. 2014. Apache cloudstack: Open source infrastructure as a service cloud computing platform. *Proceedings of the International Journal of advancement in Engineering technology, Management and Applied Science* 111 (2014), 116.

[37] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. 2017. Challenges for static analysis of java reflection-literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 507–518.

[38] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. 2014. {SAMC}: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 399–414.

[39] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. 2016. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 517–530.

[40] Haofeng Li, Haining Meng, Hengjie Zheng, Liqing Cao, Jie Lu, Lian Li, and Lin Gao. 2021. Scaling Up the IFDS Algorithm with Efficient Disk-Assisted Computing. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 236–247.

[41] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the Performance of Flow-Sensitive Points-to Analysis Using Value Flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 343–353. https://doi.org/10.1145/2025113.2025160

[42] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2013. Precise and Scalable Context-Sensitive Pointer Analysis via Value Flow Graph *(ISMM '13)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2491894.2466483

[43] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. 2017. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 677–691.

[44] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically detecting time-of-fault bugs in cloud systems. *ACM SIGPLAN Notices* 53, 2 (2018), 419–431.

[45] Jie Lu, Chen Liu, Feng Li, Lian Li, Xiaobing Feng, and Jingling Xue. 2020. CloudRaid: Detecting Distributed Concurrency Bugs via Log-Mining and Enhancement. *IEEE Transactions on Software Engineering* (2020).

[46] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1769–1786.

[47] Lindsey O'Donnell. 2019. U.S. Government, Military Personnel Data Leaked By Autoclerk. https://threatpost.com/government-military-personnel-data-leaked/149386/.

[48] Patrick Howell O'Neill. 2017. Booz Allen Hamilton leaves 60,000 unsecured DOD files on AWS server. https://www.cyberscoop.com/booz-allen-hamilion-amazon-s3-chris-vickery/.

[49] Patrick Howell O'Neill. 2017. GOP Data Firm Accidentally Leaks Personal Details of Nearly 200 Million American Voters. https://gizmodo.com/gop-data-firm-accidentally-leaks-personal-details-of-ne-1796211612.

[50] Ken Pepple. 2011. *Deploying openstack*. " O'Reilly Media, Inc.".

[51] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. Association for Computing Machinery, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

[52] JASON SILVERSTEIN. 2019. Hundreds of millions of Facebook user records were exposed on Amazon cloud server. https://www.cbsnews.com/news/millions-facebook-user-records-exposed-amazon-cloud-server/.

[53] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2011. Rolecast: finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 1069–1084.

[54] Fangqi Sun, Liang Xu, and Zhendong Su. 2011. Static Detection of Access Control Vulnerabilities in Web Applications.. In *USENIX Security Symposium*, Vol. 64.

[55] Bruce Sussman. 2020. Top 4 Types of Security Vulnerabilities in the Cloud. https://www.secureworldexpo.com/industry-news/4-types-cloud-security-vulnerability-mitigation.

[56] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: Automatically Inferring Security Specification and Detecting Violations.. In *USENIX Security Symposium*. 379–394.

[57] UpGuard Team. 2017. Cloud Leak: WSJ Parent Company Dow Jones Exposed Customer Data. https://www.upguard.com/breaches/cloud-leak-dow-jones.

[58] Keir Thomas. 2010. Microsoft Cloud Data Breach Heralds Things to Come. https://www.pcworld.com/article/214775/microsoft_cloud_data_breach_sign_of_future.html.

[59] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 1–16.

[60] Mehul Nalin Vora. 2011. Hadoop-HBase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, Vol. 1. IEEE, 601–605.

[61] Zack Whittaker. 2019. FormGet security lapse exposed thousands of sensitive user-uploaded documents. https://techcrunch.com/2019/07/25/formget-security-lapse-exposed-documents/.

[62] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. 2002. Linux security module framework. In *Ottawa Linux Symposium*, Vol. 8032. Citeseer, 6–16.

[63] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting Large-scale System Problems by Mining Console Logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 117–132.

[64] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed {Data-Intensive} Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 249–265.

[65] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. 2019. Pex: A permission check analysis framework for linux kernel. In *28th ${USENIX}$ Security Symposium ({USENIX} Security 19)*. 1205–1220.

[66] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. Lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association, Berkeley, CA, USA, 629–644.