International Journal of Automation and Computing X(X), X X, X-X
DOI: XXX

Toward Discovering Logic Flaws within MongoDB-Based Web Applications

Shuo Wen[1] Yuan Xue[2] Jing Xu[1] Li-Ying Yuan[1] Wen-Li Song[1] Hong-Ji Yang[3] Guan-Nan Si[4]

[1]Institute of Machine Intelligence, College of Computer and Control Engineering, Nankai University, Tianjin 300350, China
[2]Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, Tennessee 37212, United States
[3]Centre for Creative Computing, Bath Spa University, Bath, BA2 9BN, United Kingdom
[4]School of Information Science and Electrical Engineering, Shandong Jiaotong University, Jinan 250357, China

Abstract: Logic flaws within web applications will allow malicious operations to be triggered towards back-end database. Existing approaches to identifying logic flaws of database accesses are strongly tied to SQL statement construction and cannot be applied to the new generation of web applications that use NoSQL databases as the storage tier. In this paper, we present Lom, a black-box approach for discovering many categories of logic flaws within MongoDB-based web applications. Our approach introduces a MongoDB operation model to support new features of MongoDB and models the application logic as a Mealy finite state machine. During the testing phase, test inputs which emulate state violation attacks are constructed for identifying logic flaws at each application state. We apply Lom to several MongoDB-based web applications and demonstrate its effectiveness.

Keywords: Logic Flaw, Web Application Security, MongoDB.

1 Introduction

Web applications have become a major information access portal these years. These applications interact with back-end databases on behalf of their users. The back-end database executes all the operations requested by the web application with its privileges, and therefore the application is indispensable for ensuring security checks effective before the database accepts an operation. So web applications become one of the primary targets for malicious acquiring or manipulating the sensitive information in back-end databases. One category of attacks exploits the application's input validation mechanisms that allow malformed user inputs to be used for constructing database operations, e.g., SQL queries. The case of notorious SQL injections belongs to this type. Another category of attacks, which is referred to as state violation attacks[1], exploits logic flaws within the application. This type of attacks misleads the application into sending database operation at incorrect application states.

In contrast to input validation vulnerabilities which have received considerable attention, only limited works have been presented to address logic flaws. The key challenge comes from the fact that logic vulnerabilities are specific to intended functionality of a particular web application, hence general approaches that can be applied to all web applications require an automated way of deriving the application's intended logic or specification.

On the other hand, NoSQL databases are increasingly being employed as an alternative to traditional SQL databases. Their notable characteristics, such as flexible data models, scalable data storage, nicely support the need of web applications where the workloads are massive and data sources may not have a predefined structure. Such flexibility also brings higher risk of logic vulnerabilities into the web applications. However, to the best of our knowledge, no previous work has made efforts to address logic flaws in web applications with NoSQL database as a backend.

In this paper, we present Lom, the first systematic black-box approach which discovers logic flaws of database access within MongoDB-based web applications. The reason why we choose MongoDB is two folds: (1) According to the DB-Engines Ranking[2], the popularity of MongoDB is the top 1 among all the NoSQL databases. (2) As far as data modeling concerned, MongoDB, which has a complicated hierarchical data model, is a representative NoSQL database.

Although a few existing solutions aim to address logic vulnerabilities within web applications, the characteristics of MongoDB make their approaches not applicable for MongoDB-based web applications: (1) Identical MongoDB operations represented in distinct programming languages have various appearances. However, previous static analysis approaches[3, 4, 5], which can only address pattern-unchanged SQL queries or specific languages, can not handle the diversified MongoDB operation appearances of multiple programming languages. (2) Some black-box approaches[6, 7], which can only target the flat data model of relational databases, are not appropriate for the hierarchical and flexible data model of MongoDB. (3) Many static techniques[3, 5] require the source code of applications for analyzing, or can only be applied to specific web development languages and platforms[5, 8]. (4) A few approaches[6] need to access to server-side session information. (5) Some previous approaches[8, 9, 10] can address only one specific vulnerability and cannot be easily extended to handle other forms of logic flaws.

Regular paper
Manuscript received date; revised date
This work was supported by China Scholarship Council, Tianjin Science and Technology Committee (No. 12JCZDJC20800), Science and Technology Planning Project of Tianjin (No. 13ZCZDGX01098), NSF TRUST (The Team for Research in Ubiquitous Secure Technology) Science and Technology Center (CCF-0424422), National Key Technology R&D Program (No.2013BAH01B05), National Natural Science Foundation of China (No.61402264).
Recommended by Associate Editor Hongji Yang

By contrast, our approach supports the features of MongoDB. We explore the protocol layer to extract the MongoDB operation regardless of programming languages and introduce MPath, which is an XPath-like representation to locate each value in the hierarchical model within the MongoDB operation. In addition, our technique is designed to be general and cover many kinds of logic vulnerabilities.

The logic of a web application is modeled by a Mealy finite state machine[11] (Mealy FSM). To discover logic vulnerabilities, the intended state machine is built as a partial state machine over the expected user inputs (MongoDB operations) when users follow the navigation paths within the web application first. After that, on basis of the inferred intended Mealy machine, we generate unexpected test inputs to exploit logic vulnerabilities within the application. These test inputs are related to three categories of attacks. After producing test inputs, we send the test web requests to web applications and evaluate the outputs to discover potential logic flaws.

Our contributions are summarized as follows:

- We present a novel black-box approach for discovering logic vulnerabilities within MongoDB-based web applications. In particular, by observing the messages in the protocol layer, our approach introduces a MongoDB operation model to represent the MongoDB actions triggered within the web application. We characterize the logic flaws over the Mealy FSM, systematically utilize the observed user inputs for deriving the specification and generate test inputs to exploit vulnerabilities.

- Our approach is able to cover numerous categories of logic flaws without the need of application source code and server-side session information, therefore it can support different coding languages and environments.

- We implemented a prototype system Lom and demonstrate that Lom can be used to identify logic flaws in today's MongoDB-based web applications.

The rest of this paper is organized as follows. We present our problem formulation in Section 3. Our approach and implementation are illustrated in details in Section 4 and Section 5, respectively. Section 6 presents our experimental results. Finally, Section 2 discusses related work and the paper is concluded in Section 7.

2 Related Works

To the best of our knowledge, only two existing researches make efforts on NoSQL database security. Okman et al.[13] analyzes the main functionality and security features of two popular NoSQL databases: MongoDB and Cassandra. Aniello et al.[14] analyzes the vulnerabilities of the gossip-based membership protocol used by Cassandra. Nonetheless, none of these approaches concentrates on the flaw with in NoSQL database based web applications, while our approach detects logic flaws with in modern MongoDB-based web applications.

Most previous researches[10, 15, 16, 17] endeavor to exploit various vulnerabilities within web applications. For instance, SecuBat[18] are used to identify input validation vulnerabilities.

Nevertheless, very few techniques address logic flaws within modern web applications. There are two categories of approaches researched for securing legacy web applications from logic flaws:

1. *Vulnerability Analysis*: It tries to identify and fix the logic vulnerabilities within the applications.

2. *Attack Detection*: It tries to detect and block logic attacks launched against the vulnerable applications.

The key issue that is common for both approaches is how to derive the application logic specification. Then the logic specification is used for either attack detection or vulnerability analysis.

The logic specification that is general to a number of web applications can be manually pre-specified. Nemesis[19] aims at providing reliable authentication and authorization mechanisms for web applications. By modifying the language runtime, it can track users' credentials and enforce pre-specified security policies over resources, such as files, database objects, etc. CLAMP[20] employs virtualization technology to isolate the application components for different users, so that the current user can only access his/her own data. However, more commonly, the logic specification is specific to each application and not available as a priori. Swaddler[1], BLOCK[21] and SENTINEL[6] establish application-specific behavioral models and identify the runtime deviation from the established model as potential logic attacks. In particular, SENTINEL focuses on securing the database access triggered by the web application based on a set of invariants extracted from execution traces. The objective of these work is to detect whether a given user input violates the application specification, while our objective is to effectively identify concrete inputs to the web application which can violate the specification, which is much more challenging.

Our work shares the same objective of identifying logic flaws within web applications as a number of existing works. Swaddler[1], WAPTEC[3], RoleCast[4], Waler[5], Doupé et al.[8], Sun et al.[12], MiMoSA[22] and FixMeUp[23] infer the logic specification from application source code, through either static analysis or instrumentation. However, these techniques are language-dependent and limited in the spectrum of logic flaws they can deal with by their capability of handling language details. For example, Waler[5] can only identity violations of value-related invariants in JSP web applications, which are inferred from dynamic executions. Sun et al.[12] assume a strong role lattice model for identifying access control flaws within PHP web applications. WAPTEC[3] collects the set of constraints along the paths leading to sensitive operations and constructs exploits to circumvent the security checks. Doupé et al.[8] specifically focus on Execution After Redirection vulnerabilities in Ruby web applications by analyzing control flows from application source code.

In contrast, our approach extracts the MongoDB operation from the protocol layer without source code requirement, and can be utilized for all programming languages

supported by MongoDB. Moreover, most of the above approaches target only one specific vulnerability and cannot be easily extended to handle other categories of logic flaws. Our technique is designed to be general and covers many kinds of logic vulnerabilities.

Techniques are also designed to discover logic flaws within web applications without source code. For example, Doupé et al.[8] and NoTamper[9] can address EAR vulnerability and parameter tampering respectively. In comparison, our approaches can cover not only these two attacks, but also forceful browsing attack. InteGuard[24] and EURECOM[25] attempt to secure multi-party web applications. LogicScope[26], SENTINEL[6] and BLOCK[21] make use of session information to construct application specifications. In comparison, our work does not require server-side session information from the application developers. Li et al.[7] proposes an automated black-box technique for identifying access control vulnerabilities. Though SENTINEL[6] and the work of Li et al.[7] can be applied to traditional RDBMS, they can not handle the hierarchical and schemaless data model of MongoDB, which brings in new challenges. Our technique supports these new features of MongoDB back-end web applications.

Web applications are more and more built with third-party web services through APIs and split at both client-side and server-side, where logic vulnerabilities might arise. Wang et al.[27] discovered logic vulnerabilities within the checkout procedures, which can be exploited by the attackers to shop for free. Its further research[28] also identified logic vulnerabilities within web-based single-sign-on services. InteGuard[24] performs security checks over a set of invariant relations among HTTP interactions to defeat logic attack at runtime. INDICATOR[29] employs hybrid analysis to infer the dependency constraints on parameters for web services. Guha et al.[30] extracted event graphs from client-side web applications and detect malicious client behaviors at runtime. Krishnamurthy[31] can be used to build secure web applications, where security policies specified by developers can be automatically verified and enforced. Our technique focuses on logic vulnerabilities within server-side web applications and has the potential to be extended to handle the above scenarios.

A number of testing tools, both open-source, e.g., Spike, Burp, and commercial, e.g., IBM AppScan, have been proposed for identifying input validation vulnerabilities within web applications[16]. They feed random inputs from a library of known attack patterns into applications. To improve the testing coverage and efficiency, random fuzzing can be enhanced by guided test input generation[17, 32, 33]. None of these technique can effectively handle logic vulnerabilities within web applications.

3 Problem Description

3.1 Background of MongoDB

3.1.1 The Data Model of MongoDB

Document In MongoDB, the basic unit of data is *document* whose structure is hierarchical and non-relational. A document includes a set of field/value pairs where the value of a field can even be a document or an *array* which is a list of values. Array values can be all the supported values for normal field/value pairs in MongoDB, even nested arrays and embedded documents. Figure 1 shows a document which employs embedded documents and array values.

```
{
  "$or" : [
            {
               "Number" : {
                             "$lt" : 10
                           }
            },
            {
               "selected" : true
            }
          ]
}
```

Figure 1 A document / A MongoDB request variable

Collection MongoDB documents are grouped as one or more *collections* in a MongoDB database. The schema of a collection does not need to be defined while the collection is created, which means users have more data-modelling flexibilities to match the design and performance requirements of an application.

3.1.2 MongoDB Wire Protocol

MongoDB offers many additional drivers for users to work with their proficient programming languages. The same operations represented in distinct drivers may have different appearances. To avoid this difference, we focus on internals of how drivers access the MongoDB server. The drivers use *MongoDB Wire Protocol*, which is a simple socket-based, request-response style and lightweight TCP/IP wire protocol, to make clients communicate with the MongoDB server through MongoDB request messages. A message defines the concrete data which an operation can access and the type of the operation. With these messages, update, delete, insert and read operations can be performed on MongoDB.

3.1.3 MongoDB Request Variable

```
struct OP_UPDATE {
  MsgHeader header;              // standard header
  int32     ZERO;                  // reserved for future use
  cstring   fullCollectionName;  // "databaseName.collectionName"
  int32     flags;                 // 0 - upsert; 1 - multiupdate
  document  querySelector;       // to select the document
  document  updateDefinition;  // to specify the update to perform
}
```

Figure 2 The structure of the update message

Figure 2 shows the structure of a category of MongoDB request message (update message). As can be seen from the figure, the data structures of most useful variables in MongoDB Wire Protocol are documents, such as the query selector and the update definition. This structure is able

to support complex commands. For instance, Figure 1 is also a MongoDB request variable (query selector). Here "$lt" is a comparison operator corresponding to "less than". Each of these document structure variables is denoted as *MongoDB Request Variable* in this paper. Apparently, all the operation parameters are placed in these hierarchical and non-relational variables.

3.2 Logic Flaws within MongoDB-based Web Applications

Figure 3 shows a simple vulnerable application to illustrate the logic vulnerabilities we concentrate on in our research. A logged in user will be redirected to the "index.php" at first. If the current user is an administrator, he is allowed to achieve links for adding new users, editing and deleting any of the registered users. If the current user is a regular user, he can only browse the page for editing his personal information.

We model a web application using a Mealy finite-state machine (Mealy FSM) model (S, s_0, Σ, Λ, T, G), where S is the set of states, $s_0 \in S$ is the initial state, Σ is the set of input symbols, Λ is the set of output symbols, $T : S \times \Sigma \rightarrow S$ is the set of transition functions mapping pairs of a state and an input symbol to the corresponding next state, $G : S \times \Sigma \rightarrow \Lambda$ is the set of output functions mapping pairs of a state and an input symbol to the corresponding output symbol.

To find out the logic flaws within a web application, we are required to analyze its two categories of Mealy FSMs:

1. *Intended FSM* (denoted as F_i), which models the behavior of the originally planned web application without any logic flaws;

2. *Realistic FSM* (denoted as F_r), which models the behavior of the actual web application implemented by the developer.

If F_r is equivalent to F_i, the web application is regarded as secure. Once disparities which involve sensitive operations exist between F_r and F_i, we affirm the application has logic flaws .

As illustrated in Figure 4, the example application has three states: the guest user who is not logged in (s_0), regular user (s_1) and administrator (s_2). Each input symbol $I \in \Sigma$ is an abstract representation of the triggered operation on back-end MongoDB (e.g. op1, op2 and op3 in Figure 3), which consists of two parts:

1. *Operation Contour* (denoted by C), which represents the contour of the operation (refer to Section 4.3.1 for details);

2. *Transmitted Parameter Mapping* (denoted by P), which represents both the parameter which can be transmitted from web request to the operation and its related value set(refer to Section 4.3.2 for details).

Each output symbol in Λ is the acceptance of the operation by back-end MongoDB.

The intended FSM (F_i) for the application works as follows: At state s_1, since it is intended that the regular user can only edit his personal information, when the regular user sends an input symbol $I_1 = C_1 \cdot P_1$, where the "userid" parameter is equal to the current user id, back-end MongoDB will accept this operation (output symbol O_1). When this user attempts to edit other users' information, delete or add a user, i.e., sending I_2 (I_2 is different from I_1 due to the diverse parameter mappings.), I_3 or I_4, MongoDB will not accept or trigger the operation (output symbol O_2).

Nonetheless, in this application, there are three logic flaws which are reflected as the discrepancies between F_i and F_r. First, the "editUser.php" fails to check whether the "userid" parameter is the same as current user's information. Second, despite the "delUser.php" checks whether current user is an administrator and seems to reject the operation from web response, it does not end the application execution, thus the back-end MongoDB operation is still triggered. Third, the "addUser.php" does not check whether the current user has the admin privilege. These vulnerabilities allow three types of attacks:

1. *Parameter Manipulation Attack*: When I_2 is sent to the application at state s_1, O_1 is returned, which means a regular user can edit other users' information.

2. *Execution After Redirection (EAR) Attack* [8]: When I_3 is sent to the application at state s_1, O_3 is returned, which means a regular user can still successfully make back-end MongoDB delete other users' information although O_2 appears to be returned from web response.

3. *Forceful Browsing Attack*: When I_4 is sent to the application at state s_1, O_4 is returned, which means a regular user can add new users.

All the attacks mentioned above are common attacks targeting different kinds of logic vulnerabilities within database based web applications. EAR attack is especially challenging due to the attack seems to be defended from web response, however, the back-end database still triggers the database operation which is not designed to run.

At a given state s, only a subset of input symbols are expected by the application (denoted as $\Sigma_{exp}(s)$) and processed to produce normal output symbols, i.e., $\Lambda_{nor}(s) = G(s, \Sigma_{exp}(s))$. The expected input symbols are the triggered MongoDB operations when the user follows the navigation links of the web application. The normal output symbols mean that MongoDB accepts the expected MongoDB operations. All the other input symbols, which are not expected at state s, should not be triggered by MongoDB, resulting in blank output symbols. A blank output symbol means that the application refuses to accept the operation and therefore back-end MongoDB does not execute anything. As shown in Figure 4, for state s_1, the expected input set is $\{I_1\}$, the normal output set and the blank output set is $\{O_1\}$ and $\{O_2\}$, respectively. For state s_2, the expected input set is $\{I_1, I_2, I_3, I_4\}$ and the normal output set is $\{O_1, O_3, O_4\}$. The behaviors of F_i and F_r over the expected input symbols should be consistent because the web application aims at implementing all the intended functionalities. Nevertheless, there may be unexpected inputs which are accepted by F_r. Therefore, if an input symbol, which is not expected at state s, can be transmitted into the application and triggered by MongoDB, MongoDB then generates

index.php

```
<?php
if ($_SESSION['privilege'] == "admin") {
    $alluser = getAllUsers();
    foreach ($alluser  as $eachuser) {
        echo "<a href = delUser.php?userid=".$eachuser.">Delete </a>";
        echo "<a href = editUser.php?userid=".$eachuser.">Edit </a>";
    }
    echo "<a href = addUser.php>Add </a>";
} else if ($_SESSION['privilege'] == "commonuser") {
    echo "<a href=editUser.php?userid=".$_SESSION['userid']."">Edit</a>";
}
                    ...
?>
```

delUser.php

```
<?php
if ($_SESSION['privilege'] != "admin")
    echo("Forbidden access.");
                    //op2
require('dbconnection.php');
$mongo = DBConnection::instantiate();
$collection = $mongo->
                    getCollection('users');
$id=$_GET['userid'];
$collection->remove(array('_id' => new
MongoId($id)));
                    ...
?>
```

editUser.php

```
<?php
                    //op1
require('dbconnection.php');
$mongo = DBConnection::instantiate();
$collection = $mongo-> getCollection('users');
$id=$_GET['userid'];
$user = array();
    $user['name'] = $_POST['name'];
    $user['password'] = $_POST['pwd'];
$collection-> update(array('_id' => new MongoId($id)), $user);
                    ...
?>
```

addUser.php

```
<?php
                    //op3
require('dbconnection.php');
$mongo = DBConnection::instantiate();
$collection = $mongo->getCollection('users');
$user =array(
        'name' => $_POST['name'],
        'password' => $_POST['pwd']);
$collection->insert($user);
echo 'User created successfully';
                    ...
?>
```

Figure 3 Example Application

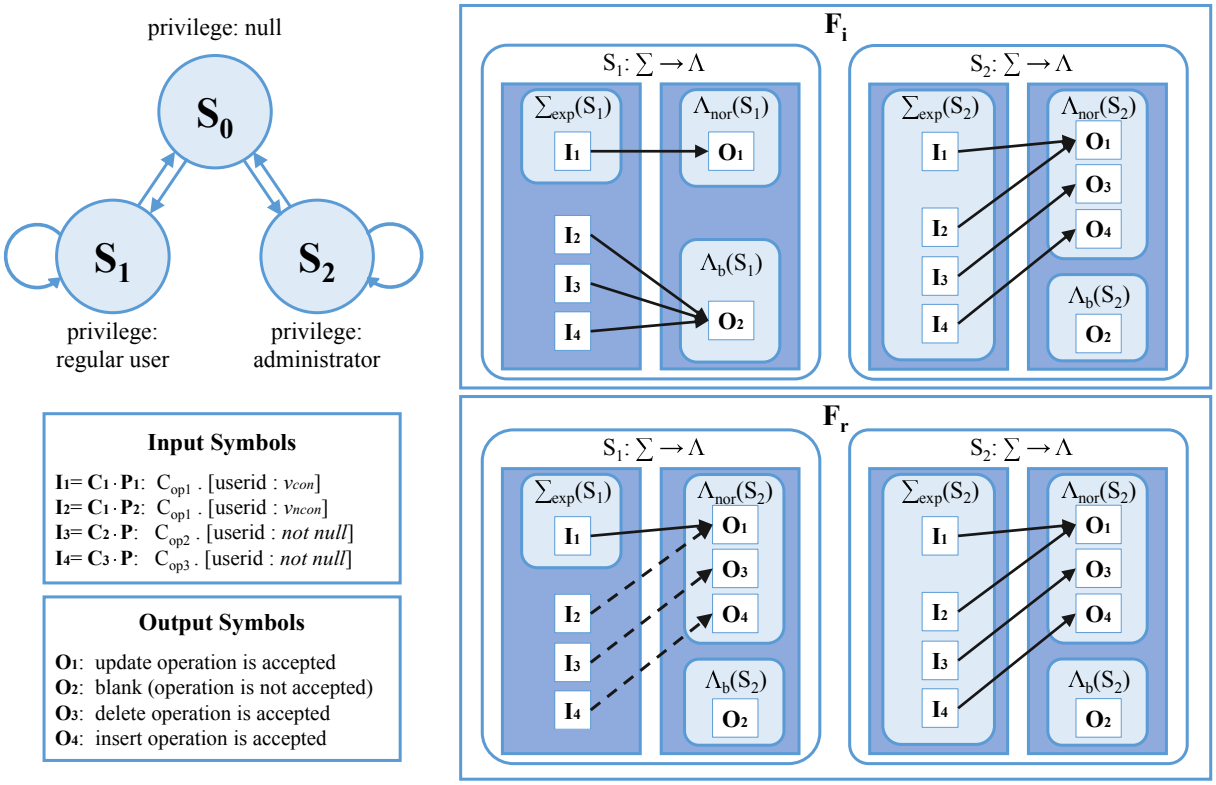

Figure 4 FSM Representation of Figure 3

an output symbol that falls beyond the blank output set, we recognize this web application has a logic vulnerability at state s. The related input symbol is defined as a *malicious input symbol* (I_{mal}).

4 Approach

4.1 Approach Overview

As mentioned in Section 3.2, we need to construct malicious inputs to verify their outputs for each state. It is a challenging task because we do not possess anything about the entire input symbol set and unexpected input symbol set at each state. Since some malicious inputs, e.g. EAR attacks, can modify the data in back-end MongoDB secretly without affecting intended web responses. To symbolize the input symbol, we need to learn the operation over MongoDB (Section 4.2). The characteristics of MongoDB make the understanding more sophisticated:

1. As illustrated in Section 3.1.2, the same MongoDB operation may have dissimilar expression in different programming language and what is more, an operation may be characterized by several statements in the source code (such as op1, op2 or op3 in Figure 3). Hence we utilize dynamic analysis but not static analysis to make our approach not constrained to specific programming language or driver. We look into the protocol layer, which is the underlying unification of distinct drivers, to extract the MongoDB operation no matter which programming language the application is written in.

2. As Section 3.1.3 shows, the basic data model of MongoDB, which is also utilized in the MongoDB request variable, is hierarchical and non-relational. MongoDB request variables are the most important components of MongoDB request messages. Thus we need to locate each field/value pair in the hierarchical data model. We present MPath to support this nested data structure.

Our approach first builds a partial Mealy FSM over the expected input domain by leveraging the collected traces. For each application, we identify user privileges and construct each privilege as a *State*. Normal users' traces are collected for different users at each state. The traces we collect include web requests/responses and MongoDB requests/responses from protocol layer. The traces are symbolized as following:

1. Input Symbolization (Section 4.3), in which we abstract concrete MongoDB operations into input symbols to profile the expected input domain at each state, i.e., $\Sigma_{exp}(s), \forall s \in S$;

2. Output symbolization, in which we observe whether the MongoDB accepts the operations or not for generating output symbols and the mappings between the expected inputs and normal output symbols, i.e., $G(s, \Sigma_{exp}(s)) \rightarrow \Lambda_{nor}(s), \forall s \in S$. Application state transitions and the corresponding input symbols that trigger the transitions are also observed in this phase, i.e., $T : S \times \Sigma \rightarrow S$.

After the inference of partial FSM, we will leverage this inferred FSM to construct unexpected inputs at each application state (Section 4.4) and test the application. Output symbols will be evaluated to discover potential logic flaws (Section 4.5).

4.2 MongoDB Operation Analysis

A MongoDB operation is related to the read, delete, update or insert message in MongoDB Wire Protocol. It can read or modify the records in MongoDB. We extract the kernel information (message/operation type, collection name and the MongoDB request variables) of a message as its MongoDB operation, which represents its execution on MongoDB performed by the user through the web application.

4.3 Input Symbolization

Given a set of MongoDB operations, we need to represent them with a finite number of input symbols. We symbolize each MongoDB operation with a two-part structure, i.e., the operation contour and the transmitted parameter mapping.

4.3.1 Variable/Operation Contour

MPath and Variable Contour Since MongoDB request variables are included in MongoDB operations, all the MongoDB request variables in the operations need to be stored reasonably. So the main challenge is how to model all of these variables in a more efficient way for convenient comparison, i.e. locating each parameter easily.

To locate each parameter, we introduce *MPath* which is an analogue of XPath. As an example, the "$lt" parameter of Figure 1 can be expressed as "$or/Number/$lt". With this kind of effective representation, we can express the original hierarchical MongoDB request variable.

We then define the *Contour* of a MongoDB request variable as the variable without any parameter values. Each original variable is represented as its extracted contour and its parameter value set. For instance, the contour of the Figure 1 can be represented as Figure 5, where "p1" and "p2" represent the value of related parameters. The top of the figure is the document view of the contour and the bottom is the MPath view which is the implementation. Both the contour and the parameter set are derived from its original variable.

Operation Contour The operation type, collection name and the variable contours of a MongoDB operation is denoted as its operation contour. Similarly, each operation is represented as its contour and its parameter value set.

4.3.2 Transmitted Parameter Mapping

We group all MongoDB operations based on their contours as well as the *kernels* of their respective web requests. A web request kernel includes HTTP method and request URL path without URL parameters. Each group is denoted as an *Operation Group*. For a MongoDB operation mo and its related web request wr, we denote a web request parameter of wr is p_{wr} and its value is $v_{p_{wr}}$, a MongoDB operation parameter of mo is p_{mo}, and its value is $v_{p_{mo}}$. If $\exists p_{wr}, p_{mo} \wedge v_{p_{wr}} = v_{p_{mo}}$ holds for all MongoDB operations and web requests within the same operation group,

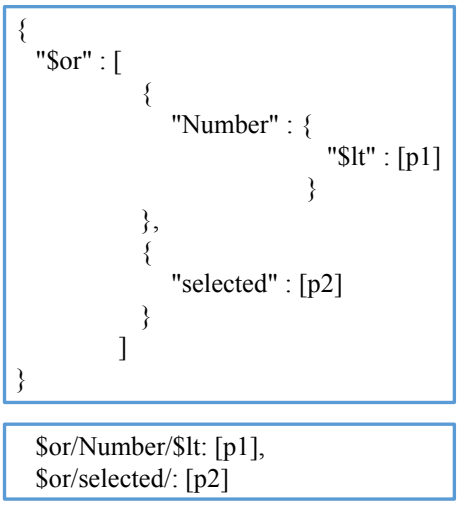

Figure 5 A variable contour (2 appearances)

we define there is a *Parameter Transmission Path* from p_{wr} to p_{mo}, and denote p_{wr} and its related value set $V_{p_{wr}}$ as a *Transmitted Parameter Mapping*.

4.3.3 Symbolization

We first profile each transmitted parameter mapping and construct this part based on its related value set, i.e., the values of all transmitted parameters because a parameter may appear infinite pairs. The characterization of each value domain is a two step process. The constraints between the parameter value set and the specific state, i.e. privilege, are extracted first by profiling each parameter at each state. For each state, the value set collected for each parameter within the same operation group is utilized for grouping the parameter into three categories:

1. *Random Parameter (denoted as $para_{ur}$)*: The value set of this type of parameter has no limitation. Its value domain is represented with two values: *null* and *notnull*.

2. *Unbounded Constrained Parameter (denoted as $para_{uc}$)*: The value set of this type of parameter is affected by certain constraints though it is infinite. Single privilege-related constraint is our focus in this paper, which means the parameter value is always specific for each user under this state (e.g., the value of "userid" of "editUser.php" at s_1 in Figure 3 is particular for each user under s_1). Its value domain is represented with three kinds of values: *null*, v_{con} and v_{ncon}, where v_{con} denotes the value satisfying a constraint linked to a specific user under this state and v_{ncon} denotes other values.

3. *Bounded Parameter (denoted as $para_b$)*: We represent its value domain with the value set and two kinds of values: *null* and v_{outb}, where v_{outb} denotes the values out of the bounded set.

We aggregate all the state views of the parameter value domains into a macroscopic view afterwards. If the value domain types of a parameter is consistent for all states, its domain type will not be changed and be recomputed. For $para_{uc}$, the updated value domain is value set divided by constraints. For $para_b$, its value domain adds additional values. If the value domain types of the parameter over different states are disparate, the more restrictive type (the restrictiveness order is defined as $para_{ur} < para_{uc} < para_b$) is adopted and its value domain will be divided. For instance, the parameter "userid" of "editUser.php" in Figure 3 is constrained by specific user at s_1, but inferred as an $para_{ur}$ at s_2. So its macroscopic type will be $para_{uc}$ and two input symbols are produced at s_2, i.e., $C_1 \cdot P_1$ and $C_1 \cdot P_2$.

4.4 Test Input Symbol Generation

As Figure 6 illustrates, there are two methods designed for generating test input symbols at a given state s.

4.4.1 Parameter Manipulation

For an expected input symbol $I = C \cdot P \in \Sigma_{exp}(s)$ at state s, we manipulate P directly, i.e., values of one or more parameters will be changed so as to make the tampered input symbol not included by the expected input set at state s. For an unbounded constrained parameter, we modify its value from v_{con} to v_{ncon}. For a bounded parameter, its value is changed to another value in the bounded set or v_{outb}. The left of Figure 6 shows an example, P_1 of input symbol I_1 is manipulated so as to generate a test input I_{mal} for s_1. This method exhibits parameter manipulation attacks, where parameter values are manipulated for violating constraints between operations and the current state.

4.4.2 Forceful Browsing/Execution After Redirection

We observe another state s' which has one or more expected input symbols excluded from the expected input set of current state s. Input symbols at s' with operation contours which are not included by the expected input symbol set of state s are chosen as test input symbols for s, i.e., $I_{mal} \in \Sigma_{exp}(s') - \Sigma_{exp}(s') \cap \Sigma_{exp}(s)$. The right of Figure 6 shows an example, the input symbols at state s_2 with C_3 and C_4 are selected as test inputs for state s_1 since they are not included by s_1. This method exhibits two types of attacks:

1. *Forceful Browsing Attacks*: One or more hidden sensitive link which should not be accessible at current state can be forcefully browsed;

2. *Execution After Redirection (EAR) Attacks*: The attacker seems to be blocked by the application from the web response of the page, but the sensitive MongoDB operations related to the page can still be successfully run on back-end MongoDB. These EAR attacks, which only manipulate the data stored in MongoDB, violate the state secretly.

4.5 Output Evaluation

We denote the output symbol generated after the test input I_{mal} being delivered into the application at state s as O_{test}. The output evaluation will determine whether O_{test} belongs to the blank output set. Since the blank output symbol means that the application refuses to accept the operation thus if back-end MongoDB trigger the operation of the test input, O_{test} falls out of the blank output set.

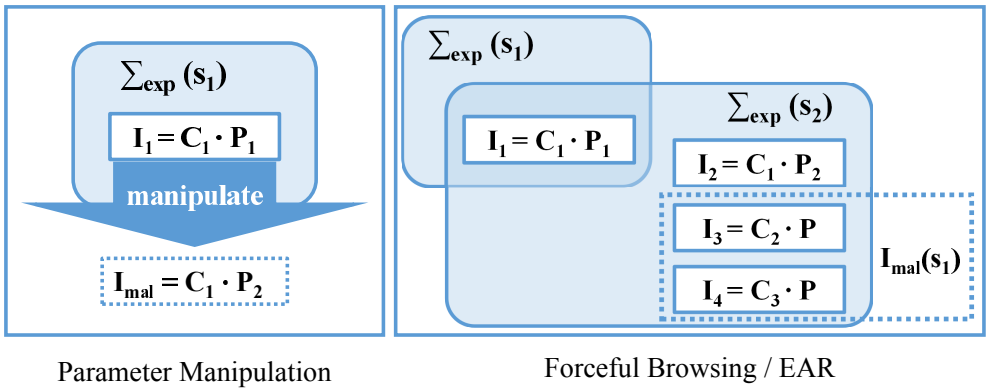

Figure 6 Test Input Generation

We collect the traces during the testing, after all the test inputs are delivered and the traces are gathered, we analyze each interaction in the traces to examine whether each test operation has been triggered or not. If a test operation is performed, we recognize its related test input as a potential logic flaw.

5 Implementation

We implement a prototype system Lom for discovering logic vulnerabilities within MongoDB-based web applications. As Figure 7 shows, Lom has three major components, including *Trace Collector*, *Specification Analzyer* and *Testing Engine*. These components are corresponding to three phases: trace collection, specification inference and testing.

5.1 Phase I: Trace Collection

Trace Collector, which collects the communication between the web application / MongoDB and the client when users navigate through the application during attack-free sessions, is implemented in our research by utilizing the open source network protocol analyzer Wireshark.

5.2 Phase II: Specification Inference

Specification Analyzer is executed in Phase II to derive both the partial Mealy FSM and the testing specification. *Symbolizer* first transforms collected traces (in Phase I) into symbolized session logs. Then, session logs are used by the *Mealy FSM Analyzer* module to derive the partial FSM, resulting in two files: *StateProfile*, which characterizes the mapping between input/output symbols at each state and *DriverSpec*, which records the transitions between the set of application states, as well as the input symbols that trigger the transitions. Finally, *StateProfile* is analyzed by *TestSpec Generator* to generate the testing specification, which includes both a set of test input symbols for each state and their related output symbols for evaluation.

5.3 Phase III: Testing

Testing Engine is executed in Phase III to test whether the application has logic vulnerabilities, based on the above derived profiles and specifications. It will produce test web requests from test inputs (by *Web Request Generator*), deliver them into the application and evaluate the test traces for logic flaw identification (by *Output Evaluator*).

Testing Controller is the core module that takes charge of the entire testing procedure. It first loads *TestSpec* and other profiles and checks the current application state. If the test of the current state is not completed, it retrieves the next available test input symbol, delegates *Request Generator* to generate a concrete web request and submit to the application. After it receives the web response, it will wrap up all the necessary information and send it to *Output Evaluator* for evaluation, where logic vulnerabilities, if exist, will be reported. If the test of the current state is completed, i.e., no test inputs are left, *Testing Controller* will move to the next available test state. It will consult *State Driver*, which loads *DriverSpec* and keeps track of the transition graph of the application, to get the path leading to the next test state. The path computed by *State Driver* is essentially the shortest path from the current state to the target state (i.e., a sequence with minimum number of input symbols), which will be instantiated by *Request Generator* and trigger the state transition step by step to the target state. This mechanism is desirable, since we cannot directly drive the application into our desired abstract state. For the example application, after we have tested state s_1 for the regular user, we have to first log out (i.e., move to state s_0) and log in as an administrator to test state s_2. If all the states have been fully tested, the testing procedure is finished.

One key challenge we need to address is how to instantiate abstract input symbols into concrete web requests with meaningful parameters. In Phase II, when we profile web requests, we also infer the value type (e.g., number, literal string) of each parameter. When *Request Generator* tries to generate the concrete value for a parameter, it checks its value type and randomly generates a value of that type or retrieve a value from a pre-loaded value store (i.e., *InputProfile*). In particular, *Request Generator* includes *Login Helper* module, which helps *Testing Engine* successfully log into the application. *Login Helper* requires the user to provide a *LoginProfile* file, which specifies the input symbol that represents the login request and at least one set of legitimate user credential, e.g., username and password, for each type of user, e.g., regular user, administrator.

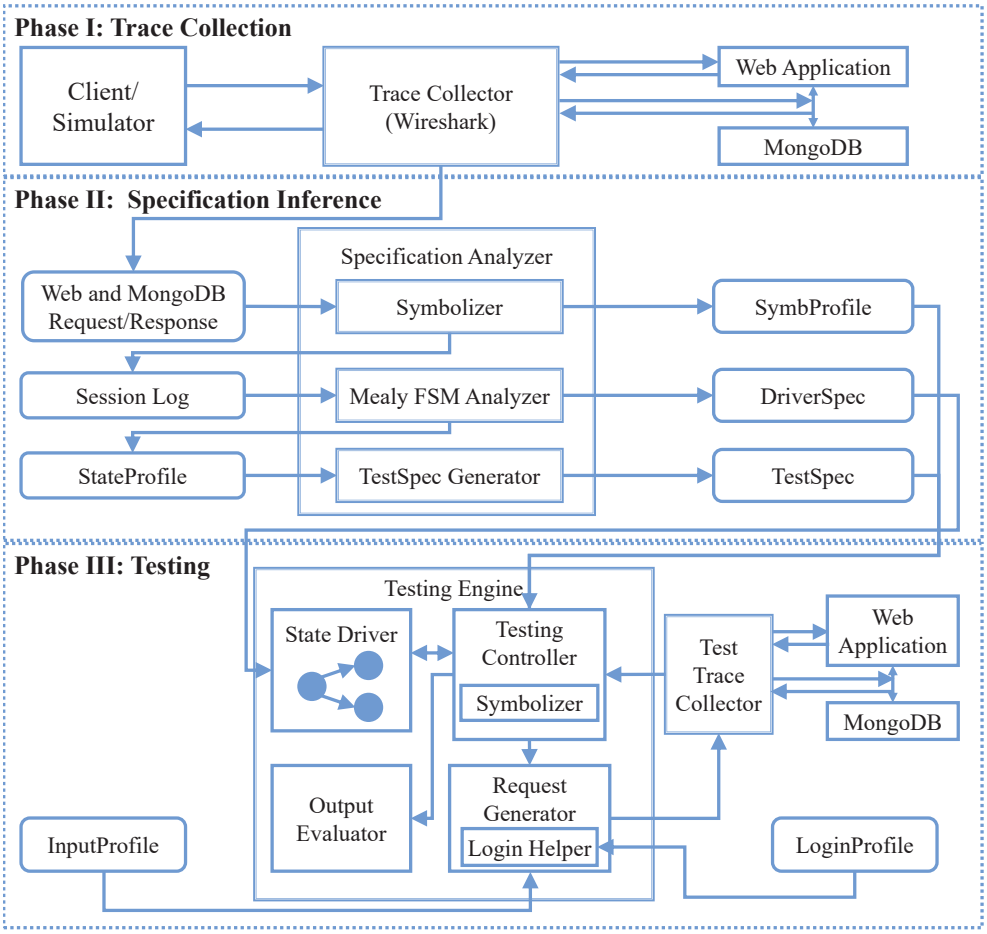

Figure 7 Prototype System Architecture

6 Evaluation

We choose a set of interactive MongoDB-based web applications for evaluating our prototype system Lom. We deploy all web applications on a 3.30GHz Intel core i3-2120 Linux server with 4GB RAM. To facilitate trace collection, we build user simulators for each application based on Selenium WebDriver. We first identify user privileges and their corresponding atomic actions by following navigation links. All of the atomic actions can be recognized as intended functions by the web application designers due to each of them follows the navigation paths under normal situation implemented by the designer. Therefore the correctness can be guaranteed. Then, the simulator performs a random sequence of atomic actions automatically with different privileges and users, each user will run all the atomic actions under his state at least once.

Our inference is performed through dynamic analysis, where the web application is executed under the constraint of navigation paths. This constraint has been applied in several existing approaches [5, 12] and shown to be effective and general to cover a large number of web applications.

6.1 Analysis of Results

Lom first runs in Phase I and Phase II to collect traces and infers the application logic specification. The statistics of collected traces and inferred FSMs are shown in Table 1, including the number of files, collected web requests, MongoDB requests, states, input symbols. Then, Lom generates the testing specification and launches the testing procedure against each web application. It constructs test web requests and sends them to the application. Testing Evaluator then evaluates the test inputs based on the collected test traces. One feature of Lom is that it also gives concrete attack vectors and evidences for further inspection.

Table 2 shows the testing results, including the number of test inputs generated by each method, flagged attacks and false positives. We also report the sum of real attacks (true positives) and vulnerable web pages. Note that these two numbers can be different, because a web page may have one or more unexpected operations which can be triggered under different states. In the following, we describe the details of logic flaws we identify from each web application. As Table 2 shows, 31 vulnerable web pages are discovered with no false positive.

Table 1 Summary of Traces and Inferred FSM

Web Application	File	Web Request	MongoDB Request	State	Input
MongoBlog	41	371	1165	2	24
QuickBlog	15	336	346	3	11
SimpleNote	21	437	493	3	10
ProductShow	8	65	25	2	2

Table 2 Summary of Testing Results

Web Application	Method	Test Inputs	Flagged Attacks	False Positives	True Positives	Vulnerable Web Pages
MongoBlog	*FE*	11	10	0	14	13
	PM	4	4	0		
QuickBlog	*FE*	21	13	0	14	10
	PM	1	1	0		
SimpleNote	*FE*	34	18	0	18	7
	PM	0	0	0		
ProductShow	*FE*	1	1	0	1	1
	PM	0	0	0		
Summary		72	47	0	47	31

FE: Forceful browsing and execution after redirection (EAR).
PM: Parameter manipulation.
True Positives: i.e. the sum of real attacks.

6.1.1 MongoBlog

There are three states in this web application: guest, regular user and admin user. Regular users can post new articles, add comments under articles, edit or delete the articles or comments created by himself. Admin users can manage all the articles and comments. Either a regular user or an admin user can mark his favorite articles. Several logic vulnerabilities are identified within this application. First, forceful browsing attacks can be applied on the application, guest users can publish and manage articles and comments as other types of users. Second, the application can be attacked by parameter manipulation, a regular user can view other user's summary page which shows articles, comments or favorite articles of corresponding user by manipulating a parameter.

6.1.2 QuickBlog

This application also has three states: guest, regular user and administrator. Only the administrator is allowed to modify all of the posts. The regular user can edit or delete his own posts. Logic flaws exist within administrative or regular users' pages which fail to check the current application state before any database operations. Thus an attacker can forcefully browse those pages and trigger sensitive operations, a regular user can perform parameter manipulation attacks to view other regular user's pages.

6.1.3 SimpleNote

There are three states in SimpleNote: regular user, user manager and super administrator. Each regular user can only view, edit and delete his own notes. User managers can manage the profile of regular users. Super administrators have the highest privilege, they are allowed to handle all users and notes. We identify logic vulnerabilities within user managers' and super administrators' pages which miss the examination of current application state. These vulnerabilities allow an attacker to browse vulnerable pages directly for managing other users' notes or profiles.

6.1.4 ProductShow

ProductShow has two states: the administrator which can add new product to MongoDB from his own page, the common user which can read products' information. An attacker can forcefully browse the administrative page due to the application does not check the current application state.

7 Conclusions

In this paper, we present the first systematic black-box approach to identify logic flaws within MongoDB-based web applications. A prototype system Lom, which introduces a MongoDB operation model to support new features of MongoDB and models the application logic as a Mealy finite state machine, is implemented and evaluated to demonstrate the practical utility of our approach.

With the development of web application technology, based on the method of this paper, there are several related areas, which will be the concentration of our further research, can be extended:

1. *The logic flaw within other NoSQL database based web application*: The NoSQL database we concentrate on now is MongoDB which is a representation of NoSQL database. Nevertheless, there are various categories of NoSQL database, e.g., columnar storage, graph storage, key-value storage, XML storage. Each kind of NoSQL database has its corresponding characteristics, these features may bring new challenges which may be worth studying.

2. *Other kinds of vulnerabilities within NoSQL database based web application*: The approach we present in this paper is target on logic flaws within NoSQL database based web applications. Meanwhile, our work solves the challenges brought by NoSQL database. It is worth

considering whether NoSQL database will bring challenges to other security problem, such as input validation vulnerabilities.

In summary, this paper makes progress on discovering logic flaws within MongoDB-based web applications and has the value of practical application. The progress also has some reference value on further research of web application security.

Shuo Wen received his B.S. in Computer Science and Technology from Nankai University, China, in 2009. Currently, he is a Ph.D. student at the Institute of Machine Intelligence, College of Computer and Control Engineering, Nankai University, China.

His research area includes networking and distributed systems with a focus on web applications and services and cloud computing. His research goal is to provide application and data integrity and privacy for next generation networking and distributed systems.

E-mail: wenshuo@mail.nankai.edu.cn (Corresponding author)

Yuan Xue received her B.S. in Computer Science from Harbin Institute of Technology, China, in 1998 and her M.S. and Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 2002 and 2005. Currently, she is an assistant professor at the Department of Electrical Engineering and Computer Science of Vanderbilt University.

Her research area includes networking and distributed systems with a focus on wireless andmobile systems, web applications and services, clinical information system and cloud computing. Her research goal is to provide performance optimization, quality of service support, application and data integrity and privacy for next generation networking and distributed systems.

Prof. Xue is a NSF CAREER Award winner.

E-mail: yuan.xue@vanderbilt.edu

Jing Xu has been a professor of Nankai University in the institute of machine intelligence since 2006.

Her research fields include software engineering, software testing and information technology security evaluation.

Prof. Xu is a member of China computer federation, software engineering technical committee.

E-mail: xujing@nankai.edu.cn

Li-Ying Yuan received her B.S. in Computer Science and Technology from Nankai University, China, in 2014. Currently, she is an M.S. student at the Institute of Machine Intelligence, College of Computer and Control Engineering, Nankai University, China.

Her research area includes software analysis.

E-mail: yuanliying@mail.nankai.edu.cn

Wen-Li Song received her B.S. in Computer Science and Technology from Nankai University, China, in 2013. Currently, she is an M.S. student at the Institute of Machine Intelligence, College of Computer and Control Engineering, Nankai University, China.

Her research area includes software analysis.

E-mail: wenli.song@foxmail.com

Hong-Ji Yang is a Professor in Centre for Creative Computing, Bath Spa University, Bath, England.

He has taken part in many important international conference, such as International Conference on Software Maintenance, the 8th IEEE Workshop on Future Trends of Distributed Computing Systems, the 26th Annual International Computer Software and Applications conference, etc. He is also the leader of Software Evolution and Reengineering Group at the Software Technology Research Laboratory.

Prof. Yang has become IEEE Computer Society Golden Core Member since 2010, also, he is a member of EPSRC Peer Review College since 2003.

E-mail: hyang@dmu.ac.uk

Guan-Nan Si received the PhD degree from Nankai University, Tianjin, China, in 2011. He is currently an assistant professor of Shandong Jiaotong University.

His research interests are software engineering and software evaluating technology.

E-mail:siguannan@163.com

References

[1] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An approach for the anomaly-based detection of state violations in web applications," in *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, RAID'07, pp. 63–86, 2007.

[2] DB-Engines Ranking, July 2014. `http://db-engines.com/en/ranking`.

[3] P. Bisht, T. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan, "Waptec: Whitebox analysis of web applications for parameter tampering exploit construction," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pp. 575–586, ACM, 2011.

[4] S. Son, K. S. McKinley, and V. Shmatikov, "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*, OOPSLA '11, pp. 1069–1084, ACM, 2011.

[5] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pp. 10–10, USENIX Association, 2010.

[6] X. Li, W. Yan, and Y. Xue, "Sentinel: Securing database from logic flaws in web applications," in *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pp. 25–36, ACM, 2012.

[7] X. Li, X. Si, and Y. Xue, "Automated black-box detection of access control vulnerabilities in web applications," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pp. 49–60, ACM, 2014.

[8] A. Doupé, B. Boe, C. Kruegel, and G. Vigna, "Fear the EAR: Discovering and mitigating execution after redirect vulnerabilities," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pp. 251–262, ACM, 2011.

[9] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan, "NoTamper: Automatic blackbox detection of parameter tampering opportunities in web applications," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pp. 607–618, ACM, 2010.

[10] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, USENIX Association, 2012.

[11] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[12] F. Sun, L. Xu, and Z. Su, "Static detection of access control vulnerabilities in web applications," in *Proceedings of the 20th USENIX Conference on Security*, SEC'11, USENIX Association, 2011.

[13] L. Okman, N. Gal-Oz, Y. Gonen, E. Gudes, and J. Abramov, "Security issues in nosql databases," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pp. 541–547, Nov 2011.

[14] L. Aniello, S. Bonomi, M. Breno, and R. Baldoni, "Assessing data availability of cassandra in the presence of non-accurate membership," in *Proceedings of the 2Nd International Workshop on Dependability Issues in Cloud Computing*, DISCCO '13, pp. 2:1–2:6, ACM, 2013.

[15] P. Chapman and D. Evans, "Automated black-box detection of side-channel vulnerabilities in web applications," in *CCS'11: Proceedings of the 18th ACM conference on Computer and communications security*, pp. 263–274, 2011.

[16] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai, "Web application security assessment by fault injection and behavior monitoring," in *WWW'03: Proceedings of the 12th international conference on World Wide Web*, pp. 148–159, 2003.

[17] M. Martin and M. S. Lam, "Automatic generation of xss and sql injection attacks with goal-directed model checking," in *USENIX'08: Proceedings of the 17th conference on USENIX Security symposium*, pp. 31–43, 2008.

[18] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "Secubat: a web vulnerability scanner," in *WWW'06: Proceedings of the 15th international conference on World Wide Web*, pp. 247–256, 2006.

[19] M. Dalton, C. Kozyrakis, and N. Zeldovich, "Nemesis: preventing authentication & access control vulnerabilities in web applications," in *USENIX'09: Proceedings of the 18th conference on USENIX security symposium*, pp. 267–282, 2009.

[20] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig, "CLAMP: Practical prevention of large-scale data leaks," in *Oakland'09: Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[21] X. Li and Y. Xue, "Block: A black-box approach for detection of state violation attacks towards web applications," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pp. 247–256, ACM, 2011.

[22] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna, "Multi-module vulnerability analysis of web-based applications," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pp. 25–35, ACM, 2007.

[23] S. Son, K. S. McKinley, and V. Shmatikov, "Fix me up: Repairing access-control bugs in web applications.," in *Network and Distributed System Security (NDSS) Symposium*, NDSS 13, February 2013.

[24] L. Xing, Y. Chen, X. Wang, and S. Chen, "Integuard: Toward automatic protection of third-party web service integrations," in *Network and Distributed System Security (NDSS) Symposium*, NDSS 13, February 2013.

[25] G. Pellegrino and D. Balzarotti, "Toward black-box detection of logic flaws in web applications," in *Network and Distributed System Security (NDSS) Symposium*, NDSS 14, February 2014.

[26] X. Li and Y. Xue, "Logicscope: Automatic discovery of logic vulnerabilities within web applications," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pp. 481–486, ACM, 2013.

[27] R. Wang, S. Chen, X. Wang, and S. Qadeer, "How to shop for free online - security analysis of cashier-as-a-service based web stores," in *Oakland'11: Proceedings of the 32nd IEEE Symposium on Security and Privacy*, 2011.

[28] R. Wang, S. Chen, and X. Wang, "Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services," in *Oakland'12: Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pp. 365–379, 2012.

[29] Q. Wu, L. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei, "Inferring dependency constraints on parameters for web services," in *Proceedings of the 22nd international conference on World Wide Web*, WWW '13, pp. 1421–1432, 2013.

[30] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for ajax intrusion detection," in *WWW'09: Proceedings of the 18th international conference on World Wide Web*, pp. 561–570, 2009.

[31] A. Krishnamurthy, A. Mettler, and D. Wagner, "Fine-grained privilege separation for web applications," in *WWW'10: Proceedings of the 19th international conference on World Wide Web*, pp. 551–560, 2010.

[32] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pp. 199–209, 2009.

[33] P. Saxena, S. Hanna, P. Poosankam, , and D. Song, "Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications.," in *NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium*, 2010.