Towards SQL Injection Attacks Detection Mechanism Using Parse Tree

Tsu-Yang Wu[1,2], Jeng-Shyang Pan[1,2], Chien-Ming Chen[1,2], and Chun-Wei Lin[1,2]

[1] Shenzhen Graduate School, Harbin Institute of Technology, Shenzhen, 518055, China
[2] Shenzhen Key Laboratory of Internet Information Collaboration, Shenzhen, 518055, China
{wutsuyang,jengshyangpan,chienming.taiwan}@gmail.com, jerrylin@ieee.org

Abstract. With the development of network technology, database-driven web applications (apps) provide flexible, convenient, available, and various services for users. User can send requests to these web apps by using browser over the Internet to get services such as e-commerce services, entertainments, and financial services. Though web environments have several advantages, various security threats have been described. Among these threats, SQL injection attack (SQLIA) is one of the most serious threats. SQLIA is a code injection attack that exploits secure vulnerabilities consisting in source codes to attack databases. SQLIA allows attackers to bypass authentication, access private information, modify data, and even destroy databases. Since many sensitive and confidential data stored in database must be kept private and secure, a mechanism to detect SQLIAs for web environments is necessary. In this paper, we define a framework named DSD (Dynamic SQLIAs Detection) to counter SQLIAs in web environments. Then, a concrete detection mechanism based on DSD is proposed to detect SQLIAs by using parse tree. The experimental results are demonstrated that our mechanism has higher accuracy, lower false positive rate, and false negative rate.

Keywords: SQL injection attacks, parse tree, detection, web environments.

1 Introduction

With the development of network technology, web app can be accessed by using browser over the Internet. Database-driven web apps can supply more flexible, convenient, available, and various services for users and they have become the most important business model for companies in various fields. Especially, user can send requests to these web apps for getting services such as e-commerce services, entertainments, and financial services. In the past, some secure mechanisms for the Internet and database [6,11,17,23,24,28,29] had been published.

Though web environments have several advantages, various security threats have been described. Among these threats, SQL injection attack (SQLIA) is one of the most serious threats for web apps [7,8,9,12]. An SQLIA which is an attacker inserts new SQL keywords or operators into an SQL query. With the altered query, an attacker can bypass authentication, obtain privacy information of users, modify or even destroy database.

Up to now, various methods to address SQLIAs for web applications have been proposed. Static analysis methods [4,10,21,25] attempt to prevent SQLIAs by finding all vulnerabilities before applications are deployed. These methods have no run time overhead. However, they analyze the source code of applications. It means that this kind of methods have two constraints. First, these methods are very host-language-specific; therefore, they cannot detect all kinds of SQLIAs. Second, these methods require to access source codes.

The mechanism Sania [20] detects SQLIAs in web applications during the development and debugging phases. The concepts of Sania are to capture an SQL query and generate an attack based on the syntax of potentially vulnerable spots in the captured SQL query. Then Sania can determine whether this query contains SQLIA flaws by comparing the parse trees of the intended SQL query.

Static and dynamic analysis mechanisms [13,14,22] have two phases, static analysis phase and dynamic analysis phase. It compares generated queries with normally expected queries. In the static phase, it analyzes a web applications source code to build models of legitimate queries. In the dynamic phase, queries are intercepted at run time and checked for conformity to the expected queries. Queries that do not match are rejected. Since this method requires to access source codes, it has restrictions when the source codes cannot be achieved. Moreover, the performance of this method depends on the quality of models built in static analysis phase.

Taint tracking mechanisms [2,3,5,16,26] attempt to solve the problem of SQLIAs by tracking user input and verifying that the input does not modify queries. However, this kind of method normally has additional requirements. The research [3,5]need to rewrite source codes to provide SQLIA detection. Other methods [2,16] require extra libraries to implement their design.

Several researchers [18,19,27] utilize machine learning technologies to detect SQLIAs. A detection method based on machine learning normally has two phases, learning phase and classification phase. In the learning phase, it utilizes a training set to build detection models. In classification phase, it judges if the query is an SQLIA with the models. The quality of training set will influence the performance of these methods.

Although several countermeasures of SQLIAs for web apps have been discussed, it still exist some drawbacks such as rewriting source codes of applications. In this paper, we first define a framework named DSD (Dynamic SQLIAs detection) to counter SQLIAs in web environments. Based on DSD, we propose an SQLIAs detecting mechanism by using parse tree. The main advantage of proposed mechanism is that it doesnt require to access the apps source code. Besides, DSD can be directly and easily embedded to existing web environments. Experimental results are demonstrated that our mechanism has higher accuracy rate, lower false positive rate, and lower false negative rate when detecting SQLIAs.

The rest of this paper is organized as follows. A framework DSD and a concrete detection mechanism are proposed in Section 2. In Section 3, we demonstrate the experimental results and conclusions are drawn in Section 4.

2 The Proposed Mechanism

In this section, we first define a framework named DSD (Dynamic SQLIAs detection) to counter SQLIAs in web environments. Based on this framework, we propose an SQLIAs detection mechanism. Notations used in this section are listed in the following:

- $G_r(\cdot)$: A function used for getting run-time stack, $G_r(\cdot) : q \rightarrow G_r(q)$, where $G_r(q)$, where q is a query.
- $G_t(\cdot)$: A function used for getting parse tree, $G_t(\cdot) : q \rightarrow G_t(q)$.
- H_c : A one-way hash function, $H_c : \{0,1\}^* \rightarrow \{0,1\}^k$.
- H_{id} : A one-way hash function, $H_{id} : \{0,1\}^* \rightarrow \{0,1,\ldots,n-1\}$, where n is an integer and depends on the number of slots in repository.
- C_t : A compressed parse tree, $C_t = H_c(G_t(q))$.
- C_r : A compressed run-time stack, $C_r = H_c(G_r(q))$.

2.1 DSD

The DSD consists of five units: Collector$_1$, Collector$_2$, Repository$_1$, Repository$_2$, and SQLIAs Agent and is depicted in Fig. 3.

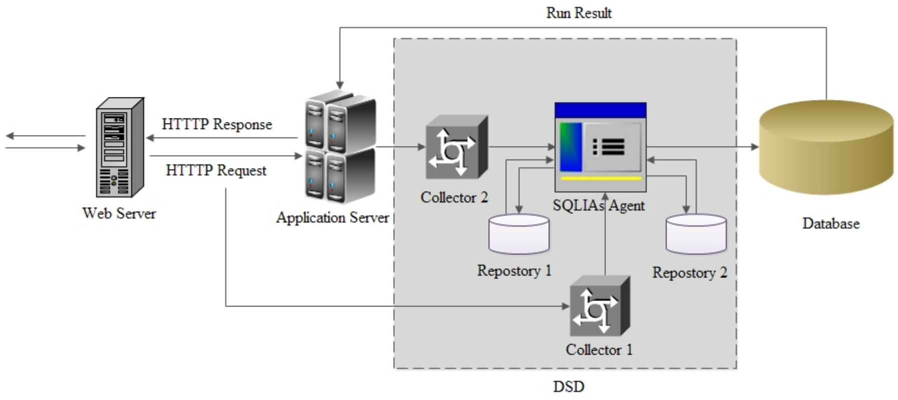

Fig. 1. The proposed framework DSD

DSD deployed between app server S and database is responsible to detect SQLIAs for a web app. The rough detecting process in DSD is described as followings.

1. User sends an HTTP request to a web application (app) within the server S. Then, the app dynamically generates a query q for this request. In this moment, Collector$_1$ captures HTTP request and sends it to SQLIAs Agent.
2. S sends q to DSD. Collector$_2$ captures some information which contains q and its run-time stack. Then, SQLIAs Agent interacts with Repository$_1$ and Repository$_2$ to verify q whether it is an SQLIA or not. If q is identified as an SQLIA, the agent will discard it. Otherwise, the agent will sent the query to the database.
3. The database executes q and sends back the result to the app.
4. The app generates a response to the user according to the result.

Note that Repository$_1$ and Repository$_2$ are two repositories. The structure of two repositories is depicted in Fig. 4. It contains a hash function H_{id} and an array A, where H_{id} is used to map a key key to the $H_{id}(key)^{th}$ slot of A. There are two operations called insertion and retrieve in the two repositories. In Repository$_1$, the insertion operation inserts 1 into $H_{id}(address)^{th}$ slot of A, i.e. $key = address$ and $value = 1$, and the retrieve operation gets 1 from the $H_{id}(address)^{th}$ slot of A. In Repository$_2$, the insertion operation inserts C_t into $H_{id}(C_t||C_r)^{th}$ slot of A, i.e. $key = (C_t||C_r)$ and $value = C_t$, and the retrieve operation gets C_t from the $H_{id}(C_t||C_r)^{th}$ slot of A.

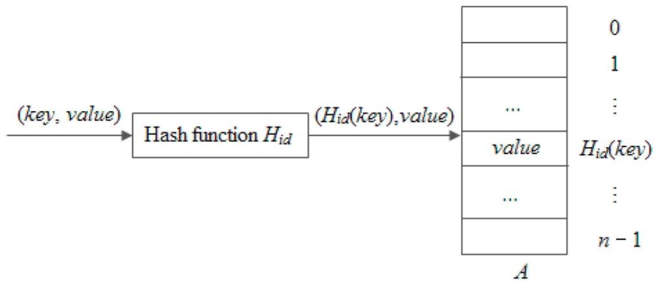

Fig. 2. The structure of repository

2.2 An Concrete SQLIAs Detection Mechanism

Based on DSD, we propose an SQLIAs detection mechanism. Our mechanism consists of two phases, classification and detection phases. When a user sends an HTTP request to an app, the classification phase is involved to identify the request whether it is first time access or non-first time access. After that, the detection phase provides SQLIA detection for this app in the above two cases.

Classification Phase. When a user sends an HTTP request to an app, Collector$_1$ captures and sends this request to SQLIAs Agent. Then, the app generates a query q corresponding to the request and sends q to Collector$_2$. It computes the corresponding run-time stack $G_r(q)$ of q and then sends q (original query) and $G_r(q)$ to

SQLIAs Agent. Upon receiving the information from Collector$_1$ and Collector$_2$, SQLIAs Agent obtains corresponding address and parameters from the request. Meanwhile, the agent parses the query q into a parse tree $G_t(q)$ and compresses it into a compressed parse tree $C_t = H_c(G_t(q))$. Then, SQLIAs Agent computes an index value $index = H_{id}(address)$ and then retrieves $A[index]$ in Repository$_1$. If the result equals to 1, the agent identifies the request as non-first time access and then invokes the non-first time access (NFTA) algorithm in the detection phase. Otherwise, the agent identifies the request as first time access and then invokes the first time access (FTA) algorithm in the detection phase. The detailed procedures of the classification phase are listed as follows.

1. Get address and parameters from an HTTP request. Note that address and parameters can be obtained in servlet program.
2. Get run-time stack $G_r(q)$ for query q. Note that the run-time stack can be implemented using high level languages such as Java.
3. Get parse tree $G_t(q)$ for query q. A parse tree can be implemented with open source tools.
4. Compute $C_t \leftarrow H_c(G_t(q))$.
5. Compute an index value $index$, where $index \leftarrow H_{id}(address)$.
6. Retrieve $value \leftarrow A[index]$ in Repository$_1$.
7. Compare $value$ with 1. If $value$ equals to 1, the NFTA algorithm is invoked. Otherwise, the FTA algorithm is invoked.

Detection Phase. In this phase, there are two cases which are the first time access and the non-first time access. The detail descriptions of the two cases are proposed as follows.

[First Time Access]
When an HTTP request is identified as first time access, SQLIAs agent inserts 1 into $H_{id}(address)^{th}$ slot of A in Repository$_1$. Meanwhile, the agent replaces all parameters of the query q with valid string such as "valid value" and obtains a transformed query q'. Note that q' is abstract valid and cannot be led to SQLIA. Then, SQLIAs agent parses the new query q' into a parse tree $G_t(q')$ and compresses it into a compressed parse tree $C'_t = H_c(G_t(q'))$. The agent compares the two compressed parse trees C_t with C'_t. If the both trees are equal, it means that the original query q is valid. In other words, the HTTP request is not an SQLIA. The original query is sent to the database. In this moment, SQLIAs agent compresses the run-time stack $G_r(q)$ of q and inserts the parse tree C_t into $H_{id}(C_t||C_r)^{th}$ slot of A in Repository$_2$. Otherwise, SQLIAs agent identifies q as SQLIAs and records it. The details procedures of FTA algorithm are listed as follows.

1. Insert 1 into slot $A[H_{id}(address)]$ in the Repository$_1$.
2. Get query q' by removing parameters from q.
3. Get parse tree $G_t(q')$ for query q'.
4. Compute $C'_t \leftarrow H_c(G_t(q'))$.
5. Compare with C_t with C'_t.

6. If $C_t \oplus C_t'$ equals to 0
   (a) q is a valid query.
   (b) Compute $C_r \leftarrow H_c(G_r(q))$.
   (c) Compute an index value $index$, where $index \leftarrow H_{id}(C_t||C_r)$.
   (d) Insert C_t into slot $A[index]$ in Repository$_2$.
7. Otherwise, q is identified as an SQLIA.

[Non-first Time Access]
When an HTTP request is identified as non-first time access, SQLIAs agent compresses the run-time stack $G_r(q)$ for original query q, ie. $C_r = H_c(Gr(q))$. Then, the agent retrieves old record C_t' in Repository$_2$ and compares the two compressed parse trees C_t' with C_t. If the both trees are equal, it means that the original query q is valid. In other words, the HTTP request is not an SQLIA. The original query is sent to the database. Otherwise, SQLIAs agent executes the procedures 2 to 7 of the FTA algorithm. The details procedures of NFTA algorithm are listed as follows.

1. Compute $C_r \leftarrow H_c(G_r(q))$.
2. Compute an index value $index$, where $index \leftarrow H_{id}(C_t||C_r)$.
3. Retrieve $C_t' \leftarrow A[index]$ in Repository$_2$.
4. Compare C_t with C_t'.
5. If $C_t \oplus C_t'$ equals to 0, q is a valid query.
6. Otherwise
   (a) Get query q' by removing parameters from q.
   (b) Get parse tree $G_t(q')$ for query q'.
   (c) Compute $C_t' \leftarrow H_c(G_t(q'))$.
   (d) Compare with C_t with C_t'.
   (e) If $C_t \oplus C_t'$ equals to 0
      i. q is a valid query.
      ii. Compute an index value $index$, where $index \leftarrow H_{id}(C_t||C_r)$.
      iii. Insert C_t into slot $A[index]$ in Repository$_2$.
   (f) Otherwise, q is identified as an SQLIA.

3 Experimental Results

In this section, we propose the experimental results for our mechanism. The experiments run on two standard PCs, PC$_A$ and PC$_B$. Both PC$_A$ and PC$_B$ have same hardware, where the processor is Intel(R) Core(TM) i5-2400M with 3.10 GHz, the RAM is 8GB, the hard disk is 500 GB, and the operating system is Windows 7. They are in the same local area network and can access each other. The architecture of experimental environment is shown in Fig. 5. PC$_B$ is used to simulate user (client) who can send accesses to related applications. We deploy one Tomcat 6 as web server and application server and MySQL 5.5 as database into PC$_A$. Then, the test application is deployed in the application server and all components of DSD are deployed in PC$_B$.

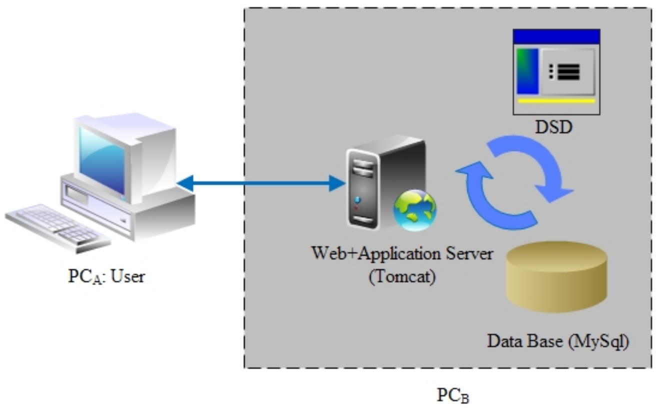

Fig. 3. Experimental environments

To demonstrate the precision and validity of our mechanism, we select four types of typical applications as test applications which are vulnerable to SQLIAs. The test applications consist of Employee Directory, Bookstore, Classifieds, and Portal [1]. Then, we choose test accesses from AMNESIA testbed suite [13,15] which is a static and dynamic analysis method to detect SQLIAs [14,15]. This suit contains a broad range of potential SQLIAs and legitimate accesses. However, it also contains some invalid accesses which cannot be accessed to database in our experimental environments. Hence, we need to remove these invalid accesses from this suit. After pre-processing the test applications, we have removed those invalid accesses and then obtain a test set which has 14674 test access including 13443 SQLIAs and 1231 legitimate accesses. The test set in our experiment is shown in Table 1.

Table 1. Test set

Types of applications	Legitimate Accesses	SQLIAs	Total
Employee Directory	124	3577	3701
Bookstore	124	3143	3267
Classifieds	348	3635	3983
Portal	635	3088	3723
Total	1231	13443	14674

Then, we demonstrate the detecting accuracy rate of our mechanism in Table 2. Obviously, the accuracy of our mechanism is over 99.9% for each type of typical application.

Finally, we show the false positive rate and the false negative rate of our mechanism in Tables 3 and 4. Obviously, the false positive rate of the proposed mechanism is less than 2% for each type of applications. The reason that false positives happened is that some queries cannot be parsed by DSD.

Table 2. The detecting accuracy of our mechanism

Types of applications	Total	Faults	Accuracy Rate
Employee Directory	3701	2	99.95%
Bookstore	3267	2	99.94%
Classifieds	3983	3	99.92%
Portal	3723	3	99.92%

Table 3. The false positive rate of our mechanism

Types of applications	Legitimate Accesses	False Positive	False Positive Rate
Employee Directory	122	2	1.64%
Bookstore	122	2	1.64%
Classifieds	348	3	0.86%
Portal	635	3	0.47%

Table 4. The false negative rate of our mechanism

Types of applications	SQLIAs	Successful Detections	False Negative Rate
Employee Directory	3577	3577	0%
Bookstore	3143	3143	0%
Classifieds	3635	3635	0%
Portal	3088	3088	0%

4 Conclusion

In this paper, we have proposed a framework DSD to counter SQLIAs for web environments. Based on this framework, a concrete detection mechanism has proposed to detect SQLIAs by using parse tree. Our mechanism does not require any access to source codes of apps. It means that DSD can be applied to existing web applications directly. Experimental results show that our mechanism has high accuracy rate, low false positive rate, and low time consumption. Hence, it is an efficient SQLIAs detection mechanism for web environments. In the future, we will compare our mechanism with previously proposed mechanisms.

Acknowledgments. This work is supported by Shenzhen Peacock Project of China (No. KQC201109020055A) and Shenzhen Strategic Emerging Industries Program of China (No. ZDSY20120613125016389).

References

1. `http://www.gotocode.com`
2. Bisht, P., Madhusudan, P., Venkatakrishnan, V.: Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks. ACM Transactions on Information and System Security (TISSEC) 13(2), 14 (2010)

3. Boyd, S.W., Keromytis, A.D.: Sqlrand: Preventing sql injection attacks. In: Jakobsson, M., Yung, M., Zhou, J. (eds.) ACNS 2004. LNCS, vol. 3089, pp. 292–302. Springer, Heidelberg (2004)
4. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embeddings. In: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, pp. 3–12. ACM (2007)
5. Buehrer, G., Weide, B.W., Sivilotti, P.A.: Using parse tree validation to prevent sql injection attacks. In: Proceedings of the 5th International Workshop on Software Engineering and Middleware, pp. 106–113. ACM (2005)
6. Chen, C.M., Zheng, X., Wu, T.Y.: A complete hierarchical key management scheme for heterogeneous wireless sensor networks. The Scientific World Journal 2014, Article ID 816549, 13 pages (2014)
7. Christey, S., Martin, R.A.: Vulnerability type distributions in cve (2007)
8. Clarke, J.: SQL injection attacks and defense. Elsevier (2012)
9. Dhamankar, R., Dausin, M., Eisenbarth, M., King, J., Kandek, W., Ullrich, J., Skoudis, E., Lee, R.: The top cyber security risks. TippingPoint, Qualys, the Internet Storm Center and the SANS Institute faculty. Tech. Rep. (2009)
10. Fu, X., Lu, X., Peltsverger, B., Chen, S., Qian, K., Tao, L.: A static analysis framework for detecting sql injection vulnerabilities. In: Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), vol. 1, pp. 87–96. IEEE (2007)
11. Guo, C., Chang, C.C., Sun, C.Y.: Chaotic maps-based mutual authentication and key agreement using smart cards for wireless communications. Journal of Information Hiding and Multimedia Signal Processing 4(2), 99–109 (2013)
12. Halfond, W., Viegas, J., Orso, A.: A classification of sql-injection attacks and countermeasures. In: Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, VA, USA, pp. 13–15 (2006)
13. Halfond, W.G., Orso, A.: Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 174–183. ACM (2005)
14. Halfond, W.G., Orso, A.: Preventing sql injection attacks using amnesia. In: Proceedings of the 28th International Conference on Software Engineering, pp. 795–798. ACM (2006)
15. Halfond, W.G., Orso, A., Manolios, P.: Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 175–185. ACM (2006)
16. Halfond, W.G., Orso, A., Manolios, P.: Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. IEEE Transactions on Software Engineering 34(1), 65–81 (2008)
17. He, B.Z., Chen, C.M., Su, Y.P., Sun, H.M.: A defence scheme against identity theft attack based on multiple social networks. Expert Systems with Applications 41(5), 2345–2352 (2014)
18. Huang, Y.W., Huang, S.K., Lin, T.P., Tsai, C.H.: Web application security assessment by fault injection and behavior monitoring. In: Proceedings of the 12th International Conference on World Wide Web, pp. 148–159. ACM (2003)
19. Komiya, R., Paik, I., Hisada, M.: Classification of malicious web code by machine learning. In: Proceedings of the 3rd International Conference on Awareness Science and Technology (iCAST 2011), pp. 406–411. IEEE (2011)

20. Kosuga, Y., Kernel, K., Hanaoka, M., Hishiyama, M., Takahama, Y.: Sania: Syntactic and semantic analysis for automated testing against sql injection. In: 23th Annual Computer Security Applications Conference (ACSAC 2007), pp. 107–117. IEEE (2007)
21. Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 1–12. ACM (2005)
22. Lee, I., Jeong, S., Yeo, S., Moon, J.: A novel method for sql injection attack detection based on removing sql query attribute values. Mathematical and Computer Modelling 55(1), 58–68 (2012)
23. Lin, C.W., Hong, T.P., Chang, C.C., Wang, S.L.: A greedy-based approach for hiding sensitive itemsets by transaction insertion. Journal of Information Hiding and Multimedia Signal Processing 4(4), 201–227 (2013)
24. Lin, C.W., Hong, T.P., Hsu, H.C.: Reducing side effects of hiding sensitive itemsets in privacy preserving data mining. The Scientific World Journal 2014, Article ID 235837, 12 pages (2014)
25. McClure, R.A., Kruger, I.H.: Sql dom: compile time checking of dynamic sql statements. In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), pp. 88–96. IEEE (2005)
26. Mitropoulos, D., Spinellis, D.: Sdriver: Location-specific signatures prevent sql injection attacks. Computers & Security 28(3), 121–129 (2009)
27. Valeur, F., Mutz, D., Vigna, G.: A learning-based approach to the detection of sql attacks. In: Julisch, K., Kruegel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 123–140. Springer, Heidelberg (2005)
28. Wu, T.Y., Tsai, T.T., Tseng, Y.M.: A revocable id-based signcryption scheme. Journal of Information Hiding and Multimedia Signal Processing 3(3), 240–251 (2012)
29. Wu, T.Y., Tsai, T.T., Tseng, Y.M.: A provably secure revocable id-based authenticated group key exchange protocol with identifying malicious participants. The Scientific World Journal 2014, Article ID 367264, 10 pages (2014)