Verifying Data Integrity for Multi-Threaded Programs

Imran Pinjari[1], Michael Shin[1] and Pushkar Ogale[2]
[1]*Department of Computer Science, Texas Tech University, Lubbock, Texas, U.S.A.*
[2]*Department of Computer Science, Stephen F. Austin State University, Nacogdoches, Texas, U.S.A.*

Keywords: Integrity Breach Condition, Data Integrity, Message Communication, Malicious Code, Multi-Threaded Program.

Abstract: This paper describes the Integrity Breach Conditions (IBCs) to identify security spots that might contain malicious codes in message communications in multi-threaded programs. An attacker can inject malicious code into a program so that the code would tamper with sensitive data handled by the program. The IBCs indicate what functions might encapsulate malicious code if the defined IBC conditions hold true. This paper describes the IBCs for multi-threaded based synchronous and asynchronous message communications in which two threads communicate via message queues or message buffers. A prototype tool was developed by implementing the IBCs to identify security spots in multi-threaded programs. An online shopping system was implemented to validate the IBCs using the prototype tool.

1 INTRODUCTION

Malicious code can compromise sensitive data to impact the business of an organization. In today's world of computers and networks, there are very high chances for malicious code to be injected into the program by an insider. Insiders (Wang et al., 2020; Fadolalkarim et al., 2020; Jiang and Qu, 2020; Zhang and Li, 2020; Rozi et al., 2020) could be software engineers who have authorized access to the source code of applications. Hence, malicious code should be detected and removed during software development as proactive measures.

A security spot is a code that may change either the value of a sensitive variable in a class or sensitive data in a database. Our previous work (Ogale et al., 2018; Radhakrishnan et al., 2021) presented the integrity breach conditions (IBCs) for non-multithreaded programs to identify security spots using sensitive variables (Ogale et al., 2018) and object references (Radhakrishnan et al., 2021). This paper extends our previous work by defining the IBCs to identify security spots in multi-threaded programs, where two threads communicate via message queues or buffers asynchronously or synchronously.

Many real-world applications have been developed with multi-threads, such as web servers, database systems, networking, video, and audio processing. Message queues or buffers are frequently used in such applications for communication to handle enormous volumes of messages or data efficiently. The messages or data sent by one thread to another can be tampered with if message queues or buffers are breached by malicious codes. Also, malicious code might be hidden in threads that communicate messages or data. It is necessary to detect malicious codes that might be hidden in the message queue or buffer as well as the threads in the multi-threaded programs.

This paper describes the IBCs to identify security spots that may contain malicious code hidden in the message queues and buffers and threads in multi-threaded programs. Some methods or functions in multi-threaded programs might contain malicious code if the IBC holds for the methods or functions. The IBCs are implemented as rules in the prototype tool, which is extended from our previous tool. The prototype tool is used to validate our research to detect security spots. The identified security spots should be manually reviewed to determine whether it is a benign code or a malicious code.

This paper is organized as follows. Section 2 describes the background of our research. Section 3 describes the integrity breach conditions for the multi-threaded programs for asynchronous message communication, followed by the IBCs for synchronous message communication in section 4. Section 5 describes the prototype tool, and Section 6

Pinjari, I., Shin, M. and Ogale, P.
Verifying Data Integrity for Multi-Threaded Programs.
DOI: 10.5220/0012130000003538
In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*, pages 588-595
ISBN: 978-989-758-665-1; ISSN: 2184-2833
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

validates this research. Section 7 describes the related work, and section 8 describes the conclusions and future work.

2 BACKGROUNDS

2.1 Glossary

The definitions of the terms used in this paper are as follows:

- **Message Queue (MQ) Connector** is a queue used for asynchronous message communication between threads.
- **Message Buffer and Response (MBR) Connector** stores the messages sent by the sender thread to the receiver thread in a message buffer in synchronous message communication. The message response buffer stores the response sent by the receiver thread to the sender thread.
- **Sensitive Variable (SV)** is a variable that stores sensitive value and whose value must not be changed maliciously. An SV is exclusively a variable of a primitive type in Java.
- **Non-Sensitive Variable (NSV)** is a variable with no sensitive value.
- **Sensitive Object (SO)** is an object of a class whose value must not be changed maliciously.
- **Non-Sensitive Object (NSO)** is an object that does not contain any sensitive values.
- **Sensitive Class (SC)** is a class that contains a sensitive variable or sensitive object (SV/SO).
- **Non-Sensitive Class (NSC)** is a class that does not contain any SV/SO.
- **Reference to Sensitive Variable (RSV)** refers to a sensitive variable (SV) in a class.
- **Reference to Sensitive Object (RSO)** refers to a sensitive object (SO) in a class.
- **Reference to Non Sensitive Variable (RNSV)** refers to a non-sensitive variable in a class.
- **Reference to Non Sensitive Object (RNSO)** refers to a non-sensitive object in a class.
- **Security Spot** is code that changes either an SV/SO value in a class or sensitive data in a database.
- **Sensitive Method (SM)** is a method that contains at least one security spot.
- **Non-Sensitive Method (NSM)** is a method that does not change the value of an SV/SO in a class or the sensitive data in a database.

2.2 Approach

Fig. 1 depicts the overview of our approach in which a list of sensitive variables or objects declared in classes and sensitive data stored in a database are given as input together with the multi-threaded programs. Using the IBCs, the programs are analyzed for data integrity to identify the security spots. The Integrity Breach Conditions (IBCs) are defined to identify the security spots in multi-threaded programs. The IBCs are the criteria for determining the security spots. If a method in a program holds an integrity breach condition, then it becomes a security spot, and the method becomes a sensitive method (SM). The security spots may be either malicious or benign. Hence, security spots should be verified manually to determine whether they are malicious or benign.

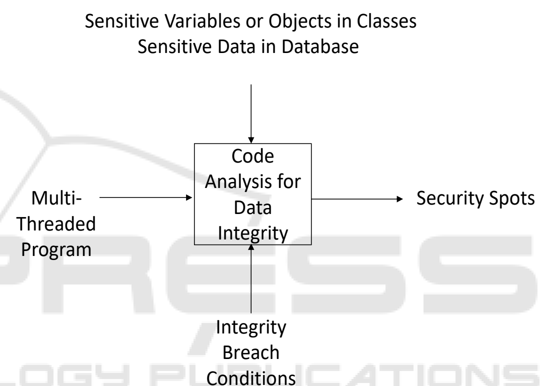

Figure 1: Overview of Our Approach.

3 INTEGRITY BREACH CONDITIONS FOR ASYNCHRONOUS MESSAGE COMMUNICATION

A message queue (MQ) connector (Gomaa, 2011) is used in asynchronous message communication between threads, which communicate messages via a message queue. An asynchronous message is sent by a sender thread to a receiver thread via an MQ connector and is stored in the MQ until the receiver reads the message. The sender thread can continue to send the next message to the receiver if the MQ is not full.

Fig. 2 depicts a message queue (MQ) connector for asynchronous message communication between the sender and receiver threads. The MQ connector provides the threads with two operations: the send() operation, called by the sender thread to store a

message in the MQ, and the receive() operation, called by the receiver thread to read a message from the MQ. The send() operation has an incoming message as a parameter, and the receive() operation returns an outgoing message read from the MQ.

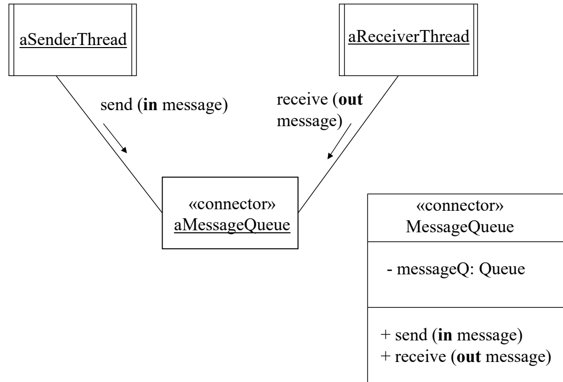

Figure 2: Message Queue Connector.

When a sender thread method (i.e., the run() function of a sender thread in Java) sends a message to a receiver thread method (i.e., the run() function of a receiver thread in Java) via a message queue (MQ) connector, the message can be an SV, NSV, RSV, RSO, RNSV or RNSO. However, RNSV is not possible in our focused programming language. Suppose a method in an MQ connector is a method M. Under these circumstances, the following IBCs are specified to identify security spots in asynchronous message communication between two threads.

IBC 1: If a sender thread method sends a message, which is either an SV or NSV, to a receiver thread method, a method M changes the message, and the receiver thread method is an SM to use the changed message, the MQ connector class becomes an SC, and the message becomes an SV in the class.

Because the message is used by an SM in the receiver thread, a change in the message will affect the SM. The message becomes an SV in the MQ connector and must not be changed in the MQ connector. As the message is stored in the MQ connector class, the class becomes an SC.

The IBC 1 can be detailed into IBC 1a, 1b, and 1c:

IBC 1a: If the message is an SV declared as a class variable in the sender thread class, the sender thread class is already an SC.

IBC 1a is trivial because it is a kind of SC definition.

IBC 1b: If the message is an NSV declared in the sender thread class, the sender thread class becomes an SC.

The message must not be changed in the sender thread as well as the MQ connector class because an SM in the receiver thread method uses the message. The message declared as an NSV in the sender thread must become an SV, and the sender thread class must become an SC.

IBC 1c: If the message is a local variable declared in the thread method, the sender thread method can be an SM.

Although the message becomes an SV, only the sender thread method can be an SM because the message is declared as a local variable in the sender thread method.

IBC 2: If a sender thread method sends a message, which is either an RSV or RSO, to a receiver thread method via an MQ connector, and a method M in the MQ connector class changes the message, the MQ connector class becomes an SC.

Because the message is an RSV or RSO in the sender thread, it must not be changed by any method in the MQ connector class. The value of the RSV or RSO in the sender thread changes if the MQ connector tampers with the RSV or RSO. Therefore, the MQ connector class becomes an SC.

Suppose the variables *a* and *b* in the Message class in Fig. 3 are SVs. The *msg* declared in the Sender class is an RSO because the variables *a* and *b* are SVs. The Message Queue class becomes an SC if the send() operation in the Message Queue class changes the value of the message *msg* sent by the Sender thread to the Receiver thread.

Under IBC 2, the following IBC 2a, 2b, 2c, and 2d are defined.

IBC 2a: The receiver thread method becomes an SM if it is an NSM and changes the value of the RSV or RSO.

When the receiver thread method changes the value of the RSV or RSO, the RSV or RSO in the sender thread method is affected. Thus, the receiver thread method becomes an SM. In Fig. 3, the Receiver thread method is an NSM and it changes the value of the variable *msg1.a* by increasing by 2000, where the *msg1* is assigned the *msg*. The Receiver thread method becomes an SM.

IBC 2b: Both the receiver thread method and method N become SMs if the receiver thread method is an NSM and calls the method N with an RSV or RSO parameter whose value is changed in the method N.

If the RSV or RSO value is maliciously changed by method N, the RSV or RSO in the sender thread method is affected. Also, the receiver thread method calls method N, which can tamper with the RSV or RSO. Both the receiver thread method and method N must be SMs. In Fig. 3, the Receiver thread method calls the method N with the message *msg1*, which changes the value of *msg1*. The method N becomes an SM. At the time, the Receiver thread method also becomes an SM if it is an NSM.

```
Class Message{
   double a; //Assume a is an SV
   double b; //Assume b is an SV
}
Class MessageQueue
{
   …
   public synchronized void send(Message message) {
      ……
      message.a = message.a - 500.00;
      ……
   }
   public synchronized Message receive(){
      …
      return message;
   }
}
Class Sender implements runnable {
   MessageQueue queue;
   Message msg;
   …
   public void run()
   {
      msg = new Message()
      msg.a = 100.00;
      msg.b = 200.00;
      queue.send(msg);
   }
}
Class Receiver implements runnable {
   MessageQueue queue;
   Message obj1;
   ……
   public void run(){
      Message msg1 = new Message();
      msg1 = queue.receive();
      msg1.a = msg1.a + 2000.00;
      msg1 = methodN(msg1);
      obj1 = msg1;
   }
   public Message methodN (Message msg) {
      msg.b = msg.b - 100.00;
      return msg
   }
}
```

Figure 3: Multi-threaded Code for Asynchronous Message Communication.

IBC 2c: A method N becomes an SM if the receiver thread method is an SM and calls the method N with an RSV or RSO parameter whose value is changed in the method N.

Method N can change the value of the RSV or RSO maliciously. This change will affect the value of the RSV or RSO in the sender thread method as well as the receiver thread method. The method N must be an SM.

IBC 2d: The class that contains the receiver thread method becomes an SC if the receiver thread method assigns the RSV or RSO to a reference to a variable or object declared in the class.

As the RSV or RSO is assigned to a reference to a variable or object declared in the receiver thread class, the value of RSV or RSO can be changed maliciously by some methods in the class. This change will affect the RSV or RSO in the sender thread method. In Fig. 3, the message *msg1* is assigned to the *obj1*, which is a reference to the Message object. The Receiver class becomes an SC.

IBC 3: Via the MQ connector class, a sender thread method sends an RNSO message to a receiver thread method that is an SM and uses the RNSO, and a method M in the MQ connector class changes the RNSO. In this case, the MQ connector class becomes an SC, and the RNSO becomes an RSO in the MQ class.

A change to the RNSO message affects the receiver thread method because the method is an SM. If a method M in the MQ connector class can tamper with the value of the RNSO, the breached RNSO is used by the receiver thread method, which is an SM. As the RNSO is stored in the MQ connector class, the MQ connector class becomes an SC, and the RNSO comes to be an RSO in the MQ connector.

The following IBC 3a and IBC 3b hold under the IBC 3:

IBC 3a: The sender thread method becomes an SM if it changes the RNSO. If the message is an RNSO declared in the sender class (but not in the method), the sender thread class becomes an SC. The RNSO becomes an RSO in the sender thread class.

Because the RNSO message is used by the receiver thread method, the change to the RNSO message in the sender thread method affects the receiver thread method, which is an SM. Thus, the sender thread method must become an SM if it changes to the RNSO. At this time, the RNSO declared in the sender thread class must be an RSO in the sender thread class, which also becomes an SC.

IBC 3b: The sender thread method becomes an SM if it changes the RNSO. If the message is a local object declared in the thread method, the sender thread method becomes an SM.

When the RNSO is declared in the sender thread method, the sender thread class cannot be an SC. However, the sender thread method must be an SM.

4 INTEGRITY BREACH CONDITIONS FOR SYNCHRONOUS MESSAGE COMMUNICATION

In synchronous message communication, a sender thread method sends a message to a receiver thread via a message buffer and response (MBR) connector (Gomaa, 2011) and waits for a reply from the receiver. When a reply arrives from the receiver, the sender can continue to work and send the next message to the receiver. The MBR connector class provides the send(), receive(), and reply() operations that enable synchronous message communication between the sender and receiver threads. A sender thread calls the send() operation to store a message in a message buffer, while the receiver thread calls the receive() operation to read a message from the buffer. The receiver thread returns a reply to the sender thread by calling the reply() operation.

Using the IBCs for asynchronous message communication, the IBCs for synchronous message communication can also be specified for both sending a message from a sender thread to a receiver thread and replying from the receiver to the sender. When a sender sends a message to a receiver, a message can be an SV, NSV, RSV, or RSO. The IBCs for asynchronous message communication are utilized for a sender to send a message to a receiver to identify SMs. Similarly, the IBCs are used to detect SMs relating to replying to the sender from the receiver. In this case, a reply can be an SV, NSV, RSV, or RSO, and the IBCs are applied reversely, meaning that a sender comes to be a receiver, and a receiver becomes a sender.

5 PROTOTYPE TOOL

A prototype tool was developed to validate the IBCs specified for multi-threaded Java programs. We extended our previous prototype tool to develop a new prototype tool for this research. The prototype tool (Fig. 4) consists of the GUI, code scanner, data integrity relation database, and data integrity analyzer.

5.1 GUI

The GUI (Fig. 4) is used to input a text file containing SVs, NSVs, RSOs, and RNSOs in a multi-threaded Java program; and a program folder encompassing the multi-threaded Java program code. The input can be entered via GUI in a format that includes the package names, class names, SVs, NSVs, RSOs, and RNSOs in each class.

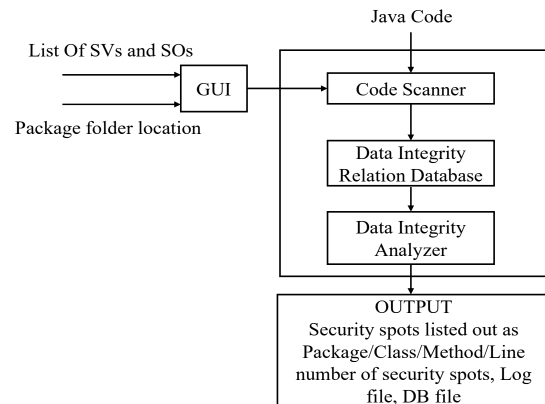

Figure 4: Prototype Tool.

5.2 Code Scanner

As our previous research selects Java code, we select a multi-threaded program in Java as our target program. The code scanner (Fig. 4) scans all classes, including the thread classes in the multi-threaded program, to build the data integrity relations used to analyze the IBCs. The class names, methods and their parameter names, and variable names are scanned and stored in the data integrity relation database. The local variables in each method are also scanned for some IBCs that are checked with the local variables. The code scanner can handle synchronized methods that communicate with each other via MQ or MBR connectors.

5.3 Data Integrity Relations Database

The data integrity relation database (Fig. 4) stores the data integrity relations extracted by the code scanner from a multi-threaded Java program. The database is a relational database containing relations handled by the SQLite3 database. The data integrity relation database contains four new data integrity relations: local variables, class objects declared at the class level, and local objects declared in methods. Also, we modified existing integrity data relations for method calls, changed variables, and used variables in each method to identify security spots in a multi-threaded Java program using the IBCs specified in this paper.

5.4 Data Integrity Analyzer

The data integrity analyzer (Fig. 4) identifies the security spots in a multi-threaded Java program using

the data integrity relation database. The data integrity analyzer detects the SMs if any method in the given program holds any specified IBC. We implemented the IBCs for both asynchronous and synchronous message communications as the data integrity analyzer to detect the security spots.

The data integrity analyzer processes all the data integrity relations in the database to identify all the SMs. When the data integrity analyzer identifies an SM, it inserts them into an output file that contains the package, class, method, message string explaining why it is SM, and line number. The data integrity analyzer repeats until all the classes in the given package are analyzed. The data integrity analyzer also creates a log file that contains information on every step executed from the beginning to the end of finding SMs.

6 VALIDATIONS

A multi-threaded online shopping system was developed to validate our approach using the prototype tool. The online shopping system consists of the Browse catalog, Make Order, Confirm Shipment and Bill Customer, Process Delivery Order, View Order, and Notify Shipment use cases. A customer can log in and browse the catalog, make an order, and view orders, while a supplier can log in to confirm shipment, bill the customer, process delivery orders, and notify the customer of shipment. All the use cases except the notify shipment are designed with synchronous message communication. The Notify Shipment use case is an example of asynchronous message communication, where the Supplier Interface sends the shipment status message to the Customer Interface asynchronously.

The IBCs specified in this paper were tested two-fold with simplified test cases for each IBC and then those for use cases in the multi-threaded online shopping system. Simplified test cases were used to check if the IBCs had been implemented correctly in the data integrity analyzer. We intentionally added at least one malicious code to the MQ and MBR connector classes or Sender and Receiver thread classes to verify each IBC. In the second round of testing, we implemented the use cases in the multi-threaded online shopping system and created the test cases for those use cases.

As there was only one asynchronous communication Notify Shipment use case in the multi-threaded online shopping system, 11 test cases were designed to test each IBC specified in this paper. We added one malicious code to the Notify Shipment use case and ran the use case 11 times separately. For comprehensive testing (11 times), the total methods recorded are 46, and the SMs identified are 7. All the SMs are correctly identified. In addition, the tool has identified 6 SCs as designed in the test cases.

The test cases for synchronous message communication were designed for Browse catalog, Make Order, Confirm Shipment and Bill Customer, Process Delivery Order, and View Order use cases. We added at least four malicious codes to the test cases for each use case. The tool has identified the unique 18 SMs and 16 SCs correctly as per the malicious code we added along with the other methods, which hold true to the IBCs specified in this paper.

7 RELATED WORK

Several methods have been suggested to examine the vulnerabilities in a program that deals with confidentiality, integrity, privacy, and availability of applications. The following discusses some of the approaches:

The Integrity Breach Conditions (IBCs) that compromise the sensitive data in an application are described in (Ogale et al., 2018). IBCs are developed using couplings associated with the language features. The authors (Radhakrishnan et al., 2021) extended the approach in (Ogale et al., 2018) by specifying IBCs by object reference (SOs and RSOs). However, the IBCs and prototype tools developed in previous work focused on general programming and did not handle multi-threaded programming. Also, local variables in methods were not handled.

The study by (Jovanovic et al., 2006) discusses the problem of vulnerable web applications by static source code analysis. This approach targets the tainted data inserted by malicious users to cause security problems. When the tainted data is used for the execution of dangerous commands, a prototype developed by (Jovanovic et al., 2006) warns about the possible vulnerabilities through data flow analysis. This approach aims at data confidentiality, integrity, and privacy.

The authors (Camps et al., 2019; Wang et al., 2020; Zhang et al., 2019) utilized machine learning to detect malicious code. The research (Camps et al., 2019) classified C source code to distinguish malicious code from benign code. In (Wang et al., 2020), the study caught malicious code based on malicious code metadata using a random forest classification algorithm. In (Zhang et al., 2019), the authors have proposed an Android malware detection

method based on the application's abstracted API calls' method-level correlation. Different behavioral patterns of malicious and benign apps are identified by combining machine learning with the detection system. Instead of machine learning techniques, our approach specified the IBCs to identify malicious code that might compromise the integrity of multi-threaded programs.

Authors (Fadolalkarim et al., 2020) proposed an anomaly detection system, AD-PROM, which would protect relational database systems against malicious application programs that steal data by tracking the calls made by application programs on data extracted from the database. The approach proposed by (Fadolalkarim et al., 2020) aims at data confidentiality, but our research focuses on data integrity in a program.

In (Jiang and Qu, 2020), the authors proposed an approach to detecting malicious code using behavior patterns identified by network behavior analysis. A memory tracking method is used to realize the real-time tracking of network behavior. The study (Jiang and Qu, 2020) focused on an outsider's attack on a network to detect malicious code. However, our approach deals with an insider's attack on a software program.

The authors (Zhang and Li, 2020) described malicious code detection using code semantic structure features to reflect semantic information. They utilized a deep learning technique with code semantic structure features to detect malicious code. In contrast, our approach used the IBCs to determine security spots that might contain malicious code.

The authors (Rozi et al., 2020) proposed a deep neural network for detecting malicious JavaScript codes by examining their bytecode sequences to protect users from cyberattacks. The study (Rozi et al., 2020) used Java bytecode, but our research used Java source code to detect malicious code.

In (Ognawala et al., 2016), the authors present a tool (MACKE) that analyzes vulnerabilities with symbolic execution and directed inter-procedural path exploration. The tool is developed using KLEE, a coverage-first symbolic execution tool for covering paths in a program. The MACKE performs a compositional analysis using symbolic execution on the functional level first and then combines the results using static code analysis based on a targeted path search. However, our tool identifies security spots using the IBCs specified for multi-threaded programs.

String analysis by (Yu et al., 2014) determines possible dangerous string constructs and provides a warning if there is a vulnerability. Malicious user input without proper input sanitization is vulnerable to attacks. String analysis focused on analyzing input strings to detect vulnerabilities in string manipulating programs. In contrast, our approach focused on the malicious code introduced by insiders in a program.

The authors (Zhioua et al., 2014) have assessed the static code analysis approaches and available tools to determine their effectiveness. The authors demonstrated that the static code analysis tools could not cover all the security issues.

8 CONCLUSIONS

This paper has described an approach to identify the security spots in multi-threaded programs that might contain malicious code. The IBCs for multi-threaded programs were specified by considering both asynchronous and synchronous messages communicated via the MQ and MBR connectors. A prototype tool was developed by extending our previous tool. The IBCs were validated with a multi-threaded online shopping system case study using the prototype tool.

We envision our future work as follows. Our future work will specify more IBCs for advanced Java language features, including an interface, inner class, and lambda expression. Also, we can extend the IBCs for smart contracts in blockchain applications, which are developed in Java or JavaScript. In addition, we can investigate artificial intelligence techniques to automatically classify benign and malicious codes in the security spots. Our approach must manually review the security spots to filter malicious codes from benign ones.

REFERENCES

Camps, G. S., Agostini, N. B., and Kaeli, D., 2019, December. Discovering Programmer Intention Behind Written Source Code. In 18th IEEE International Conference on Machine Learning and Applications (ICMLA), Florida, USA.

Fadolalkarim, D., Bertino, E., and Sallam, A., 2020, April. An Anomaly Detection System for the Protection of Relational Database Systems against Data Leakage by Application Programs. In IEEE 36th International Conference on Data Engineering (ICDE), Dallas, Texas.

Gomaa, H., 2011. Software modeling and design: UML, use cases, patterns, and software architectures. Cambridge University Press.

Jiang, C., and Qu, Q., 2020, June. A New Automatic Detection System Design of Malicious Behavior Based on Software Behavior Sequence. In 10th International

Conference on Information Science and Technology (ICIST) Lecce, Italy.
Jovanovic, N., Kruegel, C., and Kirda, E., 2006. Pixy: A Static Analysis Tool for Detecting Web Applications Vulnerabilities. In IEEE Symposium on Security and Privacy, Washington DC, pp. 258-263.
Ogale, P., Shin, M., and Abeysinghe, S., 2018, July. Identifying Security Spots for Data Integrity. In the 13th IEEE International Workshop on Security, Trust, and Privacy for Software Applications (STPSA/ COMPSAC), Tokyo, Japan.
Ognawala, S., Ochoa, M., and Pretschner, A., 2016, September. MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution. In 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore.
Radhakrishnan, R., Shin, M., and Ogale, P., 2021, July. Data Integrity Security Spots Detected by Object Reference. IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), pp 469-474.
Rozi, M. F., Kim, S., and Ozawa, S., 2020, July. Deep Neural Networks for Malicious JavaScript Detection Using Bytecode Sequences. In International Joint Conference on Neural Networks (IJCNN), Glasgow, UK.
Wang, Z., Cong, P., and Yu, W., 2020, July. Malicious Code Detection Technology Based on Metadata Machine Learning. In IEEE Fifth International Conference on Data Science in Cyberspace, Hong Kong, China.
Yu, F., Alkhalaf, M, Bultan, T., and O. H. Ibarra, O. H., 2014. Automata-Based Symbolic String Analysis for Vulnerability Detection. Formal Methods in System Design, Vol. 44.
Zhang, H., Luo, S., Zhang, Y., and Pan, L., 2019. An Efficient Android Malware Detection System Based on Method-Level Behavioral Semantic Analysis. IEEE Access, Volume: 7.
Zhang, Y., and Li, B, 2020. Malicious Code Detection Based on Code Semantic Features. IEEE Access, Volume: 8.
Zhioua, Z., Short, S., and Roudier, Y., 2014, July. Static Code Analysis for Software Security Verification: Problems and Approaches. In COMPSACW'14 Proceedings of the 2014 IEEE 38th International Computer Software and Applications Conference Workshops. Washington, USA, pp 102-109.

SCIENCE AND TECHNOLOGY PUBLICATIONS