The Journal of Systems & Software 171 (2021) 110838

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A security pattern detection framework for building more secure software

Aleem Khalid Alvi *, Mohammad Zulkernine

Queen's Reliable Software Technology (QRST) Lab, School of Computing, Queen's University, Kingston, Ontario, Canada K7L 2NB

ARTICLE INFO

Article history:
Received 16 September 2019
Received in revised form 4 September 2020
Accepted 18 September 2020
Available online 28 September 2020

Keywords:
Software design component
Security patterns
Security pattern detection technique
Security quality assurance
Secure architectural design

ABSTRACT

Security patterns are one of the reusable building blocks of a secure software architecture that provide solutions to particular recurring security problems in given contexts. Incomplete or nonstandard implementation of security patterns may produce vulnerabilities and invite attackers. Therefore, the detection of security patterns improves the quality of security features. In this paper, we propose a security pattern detection (SPD) framework and its internal pattern matching techniques. The framework provides a platform for data extraction, pattern matching, and semantic analysis techniques. We implement ordered matrix matching (OMM) and non-uniform distributed matrix matching (NDMM) techniques. The OMM technique detects a security pattern matrix inside the target system matrix (TSM). The NDMM technique determines whether the relationships between all classes of a security pattern are similar to the relationships between some classes of the TSM. The semantic analysis is used to reduce the rate of false positives. We evaluate and compare the performance of the proposed SPD framework using both matching techniques based on four case studies independently. The results show that the NDMM technique provides the location of the security patterns, and it is highly flexible, scalable and has high accuracy with acceptable memory and time consumption for large projects.

1. Introduction

The maintenance of security in a software system is a dynamic struggle. It is a battle to secure legitimate access rights from illegal use in a software system. Security patterns are always present in any security application for many decades. Since 1998, researchers have started to discover them and elaborated on the use in the security of the software system. These patterns are repetitive for providing specific security, such as "Login Window" security features. The "Login Window" is an example of a Single Access Point (SAP) security pattern, whereas "Login Window" has been in use before the discovery of the SAP security pattern. Researchers classify security patterns in the standard security pattern catalogs. These catalogs provide application details of security patterns using standard security pattern templates (Fernandez, 2013; Hafiz et al., 2011; Kienzle and Elder, 2019; Schumacher et al., 2006; Yskout et al., 2006). The discovery and documentation of security patterns are essential for designing, implementing, and enhancing the security features of software systems because of the following characteristics. Software security patterns are one of the building blocks of software security that stop attackers from illegally using a software system. Security patterns are the solution to security design problems. They are reusable security design components of a software system and provide solutions to particular recurring security problems in given contexts. They offer an easy-to-build sophisticated software security system, quickly and efficiently. Security patterns are engrafted security design knowledge from experienced security experts. Therefore, novice security developers can easily use security patterns for achieving high-quality security features in the development of a software system.

In secure software development, security patterns are used as one of the micro-architectural security components. The assurance of the correct and standard implementation of security patterns in a software system is an essential task for testing and quality assurance processes. The standard application of security patterns means that security patterns are implemented in a software system according to the detailed documentation of security patterns in available catalogs. However, the inspection of the quality of security patterns cannot be achieved without detecting the patterns in a software system. There are many software quality attributes, such as performance, modifiability, and security. The security quality attribute is specific to guard a software system providing safe execution in any device, anytime, and anyplace. However, these quality attributes affect each other; for example, the implementation of advanced security features produce the suffering of the performance of a software system. Similarly, frequent modifiability has a negative impact on security (Gorton, 2011).

* Corresponding author.
E-mail address: aleem@cs.queensu.ca (A.K. Alvi).

https://doi.org/10.1016/j.jss.2020.110838
0164-1212/

In forward software engineering, after the implementation of security requirements, the status of security patterns may be incomplete, misused, or imperfect. The testing and other quality assurance processes are utilized to determine the weaknesses and provide another chance to implement security patterns correctly. In case of any discrepancy, the source model is updated using backtracking from implementation to design and requirement phases. However, for the successfully implemented security patterns, the evaluation process may be applied to determine the level of security in a software system and how security patterns are used in combination to strengthen the security objectives (such as confidentiality) of the software system. For example, we can use more than one security pattern to improve security in the system access control architecture of a software system. Single Access Point, Check Point, and Security Session security patterns are used together to ensure the user's validity for the legitimate use of the software system resources (Fernandez et al., 2008; Schumacher et al., 2006).

In the quality assurance processes, the high-quality implementation of security patterns cannot be achieved without the detection of patterns that provide information about the implemented pattern's structure and location (Eden et al., 2018). The detection of a security pattern is the first step to measure security at the design and architecture level of a software system. The security measurement provides information for the real security expectations in a software system and gives direction for quality improvement. The detection process provides confidence to a security quality assurance (SQA) team. The SQA team checks the status of security patterns as countermeasures against security attacks in a target software system. The absence of security patterns makes security loopholes in a software system. These loopholes are usually called security flaws, defects, or weaknesses. We use the term security flaws throughout the paper. Security patterns are employed to mitigate security flaws. Security flaws invite attackers to exploit software vulnerabilities for illegitimate access to a target software system. Therefore, the detection of security patterns is the key to ensure security in a software system and support security quality attributes. It helps in the elaboration and construction phases of a secure software development lifecycle. Detecting instances of security patterns in a target system assist security developers to understand the missing security features.

The detected security patterns provide information for lost security architectural design decisions and offer help for improving and maintaining the security of a software system at a higher level. Therefore, it contributes to understanding, analyzing, and configuring a software system with specific security requirements. The detection framework is supportive of locating security patterns and understanding its orchestration. Many pattern detection research works exist, such as detection for design and anti-patterns (Dong and Zhao, 2007; Dong et al., 2008a,b, 2009; Gupta et al., 2010a,b; Pande et al., 2010c,b,a; Tsantalis et al., 2006) but only a couple of the research works are found for security pattern detection (Alvi and Zulkernine, 2017; Bunke and Sohr, 2011). However, the implemented detection methods have limitations, including process complexity, time consumption, and the determination of the location of a security pattern in the software system source code.

In this paper, we present a security pattern detection framework. The framework depends on the preprocessed data (shown as four blue shaded boxes) and three necessary techniques (shown as orange shaded boxes). The first main component is class relationship discovery. It takes two inputs, i.e., a software system model and an empty class relationship matrix. The empty class relationship matrix is developed using the class names from a software system's source code or design document. It is an adjacency matrix of a software system graph, and it is filled with relationship information among classes. The class relationship discovery process uses a software model to detect relationships among classes of the software system graph and fill the empty matrix. The class relationship matrix is also called a target system matrix. It is a logical matrix as the relationship information between the classes is stored as 0 or 1.

The second necessary component of the framework is security pattern matching. It takes two inputs, i.e., the filled class relationship that is the output from the class relationship discovery and security pattern matrix units. The security pattern matrix is the logical matrix that has information about the relationships of security pattern classes. The information about relationships among security pattern classes is extracted from security pattern documentation, as shown in Fig. 1. The security pattern matching is used to match the security pattern matrix inside the target system matrix.

In some cases, the result may be many similar security pattern matrices. The third essential part of the framework is semantic analysis. It is employed for removing confusion on similar matrices and detecting an actual security pattern matrix. The semantic analysis takes two inputs, i.e., all similar security pattern matrices and dictionary data. The dictionary data is prepared during the preprocessing stage using security pattern documents, security requirement documents and developer's intuition. The main overhead here is to find the actual security pattern from similar detected security pattern matrices depending on the dictionary data. The sophisticated and comprehensive dictionary can reduce a significant number of similar irrelevant security pattern matrices. The dictionary data does not guarantee that it can completely filter all similar security patterns and provide actual security patterns. If the dictionary partially fails, then after filtering (through dictionary), remaining pattern matrices have to be scrutinized manually for detecting an actual security pattern matrix. The fourth essential part is the detection report that provides information for the implemented security patterns and their locations inside a software system. The general overview of the proposed framework is shown in Fig. 1. The detection framework is explained in detail in Section 3.

There are many matching algorithms available for the detection of design patterns inside a target system (Dong et al., 2007, 2008b, 2009; Tsantalis et al., 2006). However, no matching algorithm has been developed for the detection of security patterns. We develop ordered and non-uniform distributed matrix matching techniques for the SPD framework (Alvi and Zulkernine, 2017). We evaluate the SPD framework using the School Semester Scheduling system (SSS). Then we extend the evaluation process to three open-source Java-based target systems, i.e., the Simple Android Instant Messaging (SAIM) client (Mermerkaya, 2013), the Automated Teller Machine Simulator (ATM sim.) (Bjork, 2019), and the Electronic Voting System (VoteBox) (Sandler et al., 2008).

We initially implemented the ordered matrix matching (OMM) technique using the SPD framework (Alvi and Zulkernine, 2017). In this paper, we update the OMM technique and improve and extend the security pattern detection framework. We also propose a novel security pattern matching technique called non-distributed security pattern matrix matching (NDMM). We develop the multi-level security pattern dictionary (MSPD) using the N-gram model of natural language processing. The discovery of security patterns gives power to security developers to orchestrate correct security architecture with a high level of security. Therefore, software systems will provide the best performance, demonstration, and perseverance of security features. In summary, we have the following main contributions in this paper:

1- We present a security pattern detection (SPD) framework that can be used as a platform for data extraction, matching, and semantic analysis techniques.

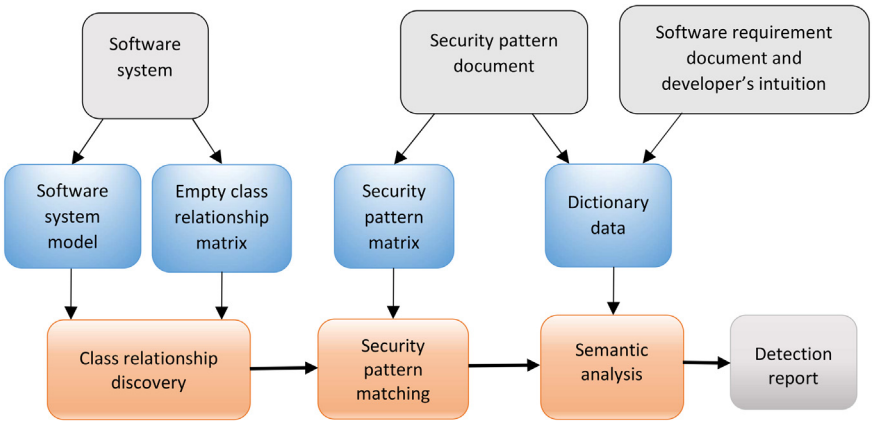

Fig. 1. Overview of the security pattern detection (SPD) framework.

2- We implement data extraction, security pattern matching, and semantic analysis techniques.
3- We discover associations of security pattern elements, to the security pattern matching and semantic analysis components of the SPD framework.
4- We apply an updated ordered matrix matching (OMM) technique and propose a novel technique based on non-distributed security pattern matrix matching (NDMM) for security pattern matching.
5- We use the N-gram model of natural language processing (NLP) for developing the multi-level security pattern dictionary (MSPD).
6- We compare and contrast the two matching techniques, OMM and NDMM, for the improved SPD framework.
7- We evaluate the SPD framework and its internal techniques using four open-source Java-based software systems.

The organization of the paper is as follows: In Section 2, we depict the background of the research work. This section includes the requirements of a pattern template and its role in security pattern detection. In Section 3, we present the security pattern detection framework in great detail. In Section 4, we explain the implementation of the proposed security pattern matching techniques in the SPD framework. Section 5 precisely elaborates on the evaluation of the detection framework and analyzes the results for the SPD framework. In Section 6, we describe the related work on pattern detection frameworks. In Section 7, we elaborate threat to the validity of the security pattern detection. Finally, in Section 8, we conclude our research work and present some directions for the future based on the limitations.

2. Security pattern template for the SPD framework

In this section, we elaborate on the background of pattern detection and explain the usefulness of a security pattern template in the implementation of the SPD framework. The development of a security pattern catalog started in 2002 (Kienzle and Elder, 2019; Kienzle et al., 2019). Many security pattern catalogs have been introduced to help developers utilize security patterns in software development processes (Fernandez, 2013; Hafiz et al., 2011; Kienzle and Elder, 2019; Kienzle et al., 2019; Slavin and Niu, 2018). They provide an enumeration of security patterns. Every catalog first introduces a template for their presented list of security patterns. Security pattern template (SPT) is the structure of pattern descriptions that express patterns formally in pattern documentation. Pattern templates evolved from the Alexandrian pattern template (Alexander et al., 1977) to the Gang of four (GoF) design pattern template (Dong and Zhao, 2007) and finally transformed into other pattern templates, such as the Schumacher's security pattern template (Schumacher et al., 2006). In security pattern documentation, the format of the pattern has not yet been standardized by any organization (Meszaros and Doble, 1997). We observe that it applies to all types of pattern templates. However, a consensus has been reached for which pattern elements should be included in the templates.

A security pattern template is a specific document that contains elaborative information of a security pattern in a particular format. Every pattern template has a unique list of pattern elements. Each pattern element has a specific role in a pattern template. The name of the pattern template gives an idea of the purpose of the whole security pattern; nevertheless, every pattern element defines its characteristics explicitly. The natural classification of security patterns was presented after a comprehensive survey of available security pattern classifications (Alvi and Zulkernine, 2011, 2012). The proposed security pattern template in the natural classification of security patterns provides structural, behavioral, and semantic information, as shown in Fig. 2. It shows the connection of the security pattern template with the natural classification of security patterns and security pattern detection framework. Some of the pattern elements are used for defining the natural classification of security pattern based on security flaws, and others help detect security patterns. The pattern elements "Security Objectives", "Related Security Flaws", and "Related Attack Patterns" are used for defining the natural classification of security patterns. The other pattern elements such as "Security Pattern Names", "Also Known As", "Static Structure", "Dynamic Structure", and the whole security pattern document are used for providing required information to the components of the security pattern detection framework. These pattern elements are employed to provide structural, behavioral, or semantic information for the framework. Some pattern templates only provide structural information under the pattern element "Structure", such as the GoF template (Dong and Zhao, 2007).

The use of security pattern elements in the detection framework is expounded as follows: The proposed framework uses the pattern element "Also Known As" and "Solution" from the security pattern template (shown as blue arrows and boxes in Fig. 2). However, for the development of a multi-level dictionary for semantic analysis, we parse the whole security pattern and security requirement documents. The detailed semantic analysis is elaborated in Section 4.3. The pattern element, "Solution", is further divided into sub-elements, such as Static Structure, Dynamics Structure, Participants, Collaborations, and Strategies. The pattern element, "Solution", is defined as a fundamental principle for a solution of a security problem and includes an exhaustive description of pattern elements with an associated mechanism.

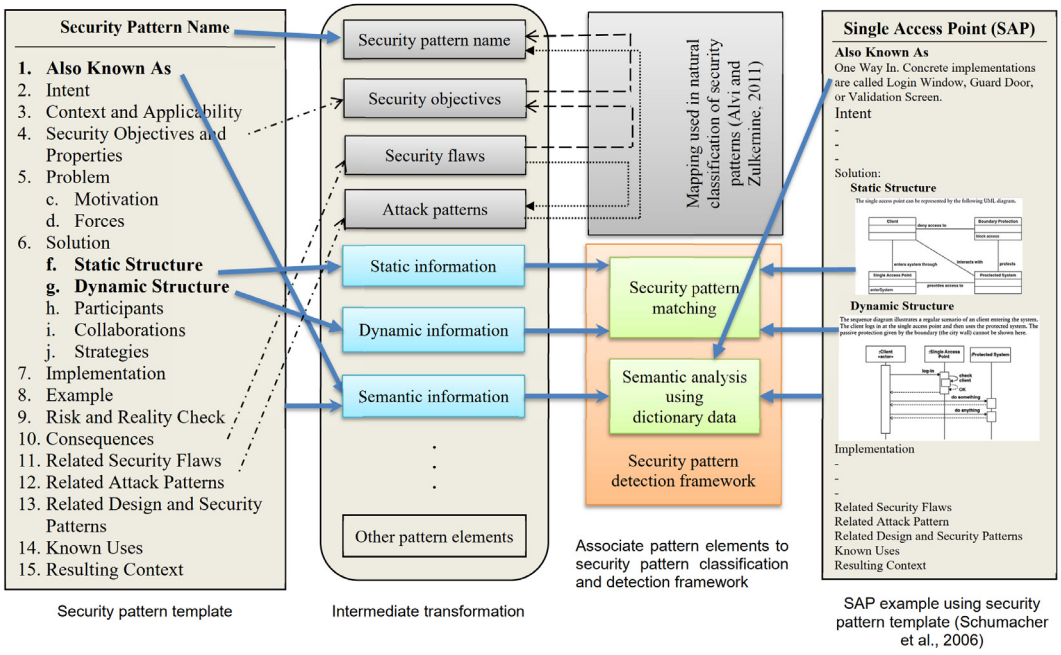

Fig. 2. Detailed security pattern template elements associated with the SPD framework.

Similarly, further sub-elements of pattern element, "Solution", are defined to provide the details of a security problem solution. For example, under the sub-element, "Static Structure", a security pattern document offers a class diagram that represents the static structure of a security pattern. The class diagram for a particular pattern contains unique information; therefore, it is used to detect the pattern on a structural level of a software system. Similarly, under sub-element, "Dynamic Structure", defines the interaction diagrams. The interaction diagrams preserve the dynamic behavior of a security pattern and employ it to help in the detection processes of a security pattern. Some detection frameworks utilize both structural and dynamic properties to reduce false positives. The pattern element, "Also Known As", is used for the semantic analysis of detected security patterns. In this paper, we use static, dynamic and semantic information of a security pattern template (SPT) for the detection framework (see the mapping of SPT to SPDF in Fig. 2).

As a concrete example, we map the security pattern template with the Single Access Point (SAP) security pattern document. The SAP organizes according to the security pattern template. A security pattern template element used in SAP security pattern called "As Known As" is available with the other names of SAP, such as One Way In. Also, the concrete implementations are called Login Window, Guard Door, or Validation Screen. A security pattern template element, "Solution", has two essential sub-template elements, i.e., "Static Structure" and "Dynamic Structure". "Static Structure" is a class diagram, and "Dynamic Structure" is a sequence diagram from the SAP security pattern document (refer to Fig. 4). However, a security requirement document is also used for semantic analysis and dictionary development. The use of the security requirement document is explained in Section 4.3.

The static and dynamic pieces of information about security patterns are the basis for developing security pattern graphs. These graphs are used to generate matrices for the matching techniques in the security pattern detection (SPD) framework. Moreover, the semantic information is used to reduce false positives. In the next section, we discuss the SPD framework and show how to employ the elements from the security pattern template (SPT) for developing security pattern graphs.

3. The detailed detection framework

3.1. Introduction

The security pattern detection (SPD) framework employs security patterns and a target software system as inputs. They process both security patterns and target software system data and detect the existence of security patterns. The fundamental structure of a security pattern detection framework includes the initial, intermediate, and final processing. The detailed framework is shown in Fig. 3. The initial setup consists of preprocessing the data to be used as inputs to the framework and will be discussed in Section 3.2. The inputs are security patterns, target system graphs, target system unified modeling language (UML) model, and dictionary data (as labeled in Fig. 3 as ①, ②, ③, and ④, respectively).

The fundamental part of the detection framework contains class relationship discovery (information extraction), security pattern matching, and semantic analysis using dictionary data. The output of the framework is a report for the information of the existence of security patterns, including their location in the software system source code. We develop a tool called the Security Pattern Detection (SPD) tool based on the proposed framework.

The representation of a graph is a matrix in linear algebra. The security pattern and target software system graphs are stored in the database as matrices. Throughout the paper, we use the word 'matrix' instead of 'graph'. A class diagram can be developed using a reverse engineering technique from a software system source code or may be produced during the designing phase of the target system. The intermediate processing consists of a class relationship discovery and security pattern matching units and will be discussed in Section 3.3. The class information extraction (CIE) subunit extracts information from the class diagram of the target system and stores it as a target system matrix (TSM), also called a class relationship graph (CRG) database.

The security pattern matrix is prepared using class and sequence diagrams available in security pattern documentation during initial processing. All security pattern matrices are stored in a security pattern graph (SPG) database. These matrices are

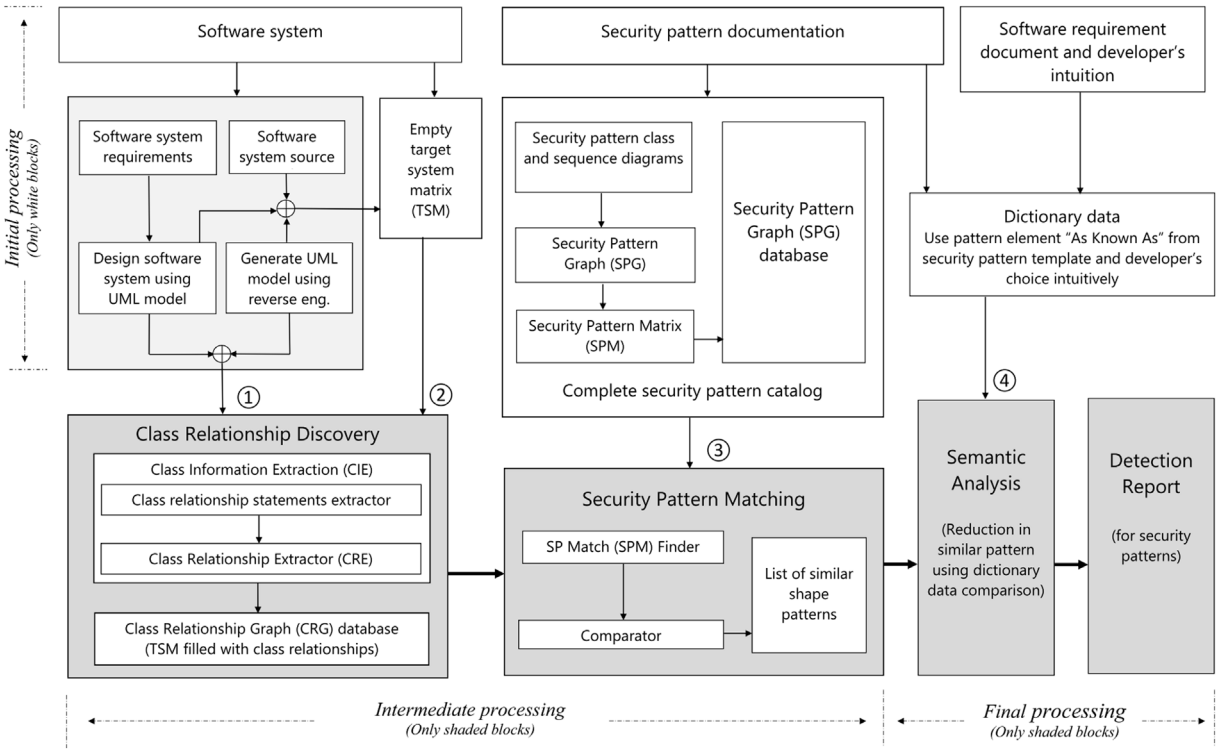

Fig. 3. The detailed security pattern detection (SPD) framework.

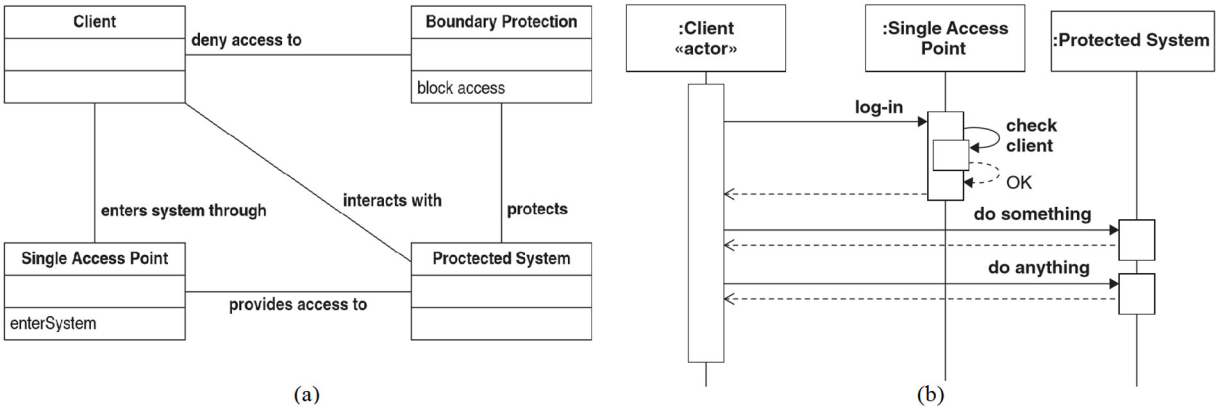

Fig. 4. SAP security pattern (Schumacher et al., 2006) (a) class and (b) sequence diagrams.

input into the security pattern matching (SPM) unit. The security pattern matching (SPM) unit detects the security pattern matrix inside the CRG database (a large target system matrix). The final processing consists of the semantic analysis and detection report generation units that will be discussed in Section 3.4. In some cases, the output from the security pattern matching (SPM) unit may be more than one security pattern matrix. To find the actual security pattern matrix, we apply semantic analysis using a data dictionary to reduce the rate of false positives. In Fig. 3, input ④ is the dictionary data for a specific security pattern. The analysis results provide information about the existence or non-existence and the location of security patterns. Finally, the SPD tool generates the detection report.

In the next subsections, we elaborate on all the processes for the detection framework that are practically implemented in the SPD tool.

3.2. Initial processing

The preparation of input files for class relationship discovery, security pattern matching, and semantic analysis units is part of the initial processing. We prepare an empty target system matrix (TSM) using the list of class names available either in the software system source code or software system model diagram.

The first input ① to the class relationship discovery unit is the software system UML model in the text file format, as shown in Fig. 3. We use Java-Editor 15.26 (2016) tool (Röhner, 2018) that saves the UML model in a text file format. For example, the information from the class diagram, including classes and their types of relationships are visible as rectangles and lines that are connecting rectangles (classes) and stores as text in the text file. The line connecting class Login and class Client in a diagram is saved in the text file.

V1 = Login#Client#AssociationDirected#1...1#provides access to#1...1#1#0#-1

We explain all the parameters used in the above line in Section 3.3. For preparing the second input ②, first, we utilize a source monitor software (Campwood Software, 2018) to extract the list of class names from the target system source code or use a software system UML model. Second, the list is used to develop an empty TSM. The names of the target system classes are listed in the same order in the first row and in the first column of the empty matrix to develop the TSM. Therefore, the second input ② to class relationship discovery unit is the empty TSM.

We prepare the software system UML model by reverse engineering a target system utilizing the Java-Editor tool, or it can be designed during the inception phase of the software development. In any of the cases, we provide the software system model as an input to the SPD tool.

The security pattern matching unit takes two inputs: one is the output from the class relationship discovery unit, and the other is the preprocessed input ③. For the preprocessed data input ③, we extract a security pattern graph from the security pattern class and sequence diagrams. Fig. 4(a) and (b) show class and sequence diagrams of the Single Access Point (SAP) security pattern, respectively, from Schumacher's catalog (Schumacher et al., 2006). The class and sequence diagrams of SAP security pattern are used to develop the SAP security pattern graph and store it as the SAP security pattern matrix (SPM). We consider the Client class as human. Therefore, the size of the SPM is 2 by 2. We prepare security pattern graphs of all security patterns using their class and sequence diagrams and store them as matrices in the security pattern graph (SPG) database.

The fourth preprocessed data input ④ is the dictionary data to the semantic analysis unit. The dictionary data is unique for every security pattern. It is prepared using the security pattern element called "Also Known As". The pattern element, "Also Known As", consists of all possible names of a security pattern. However, the developer may select something different based on their working scenario and experience. Therefore, the developer's intuition is also used as input to the data dictionary. Further, we collect and use all nouns through parsing (using N-gram model) a complete security pattern and security requirement documents as input to the dictionary data. The final stage of the dictionary has almost all the possible names (including different font cases and styles) that can predict the functionality of the corresponding security pattern.

The SPD tool employs the matrix data type for intermediate processing to detect the SPM inside the TSM. Therefore, the purpose of the initial processing is to prepare an empty target system matrix, a security pattern matrix, a target system UML class diagram, and the dictionary data of a security pattern in text format. These preprocessed inputs are used in intermediate processing and are explained in the next subsection.

3.3. Intermediate processing

The intermediate processing consists of class relationship discovery and security pattern matching units. It takes input from preprocessing data, i.e., an empty TSM, a UML class diagram in text format, and an extracted SPM. The class information extraction (CIE) subunit extracts the class relationship information from the target system using a UML system class diagram. The class relationship information is mined from the class relationship statements inside the UML model file using the class relationship extractor (CRE) subunit. The class relationship information includes relationships between classes, types of relationships, and cardinality from the UML class diagram of the target system. The standard class relationship statements are stored in the text file of the system UML class model. We use a direct relationship between classes as input, either 0 or 1 in TSM.

Now we discuss internal components of the text file of the UML software system class model. The UML system model preserves relationship information among the classes. Every relationship between two classes is called "standard class relationship statement" and is stored in the following format, where '#' sign is used as a separator.

class relationship label = class relationship starts from # class relationship ends to # relationship name # cardinality (from) # explain relationship as stereotypes in text # cardinality (to) # recursive association # connection is turned ON/OFF # connection is edited

The following excerpt of a class relationship statement is used to explain the above statement:

V0 = Login#Admin#AssociationDirected#1...1#provides access to#1...1#1#0#-1

Each item of the standard class relationship statement is explained below using the above example:

1. class relationship label : the unique label to identify the relationship, e.g., '*V0*'
2. class relationship starts from : the first-class name from where the relationship originates, e.g. '*Login*'
3. class relationship ends to : the second-class name where the relationship ends, e.g., '*Admin*'
4. relationship name : the name of the relationship, e.g., '*Association Directed*'
5. cardinality (from) : the cardinality of the relationship from the first-class, e.g., '*1...1*'
6. stereotypes : defines the reason for the relationship in text, e.g., *provides access to*
7. cardinality (to) : the cardinality of the relationship from the second-class, e.g., '*1...1*'
8. recursive association : '1' then recursive relationship exists; otherwise, it is 0, e.g., '*1*'
9. connection is turned ON/OFF : '1' if turned ON; otherwise '0', e.g., '*0*'
10. connection is edited : '0' if not edited and otherwise '−1', e.g., '*−1*'

The class-to-class relationship is extracted using the class information extraction (CIE) subunit and stored it into an empty TSM. The filled TSM with class relationship information represents the class relationship graph (CRG) database, as shown in Fig. 5.

We observe that the classes of the Simple Android Instant Messaging (SAIM) target system are inserted in the first row and column in the same order, and the digit '1' represents the existence of a relationship among classes; otherwise, it is '0'. The final step of the intermediate processing is to match the security pattern matrix (SPM) of the SAP security pattern inside the target system matrix (TSM) of the SAIM target system. The security pattern matching unit uses a matching matrix algorithm. The SPM unit has an SP Match Finder and Comparator subunits. The SP Match Finder subunit detects a security pattern and then compares it using a comparator unit with an SPM for error checking. The output result is the similar shape of a SPM matrix. However, some of the outcomes are false positives. The detailed matrix matching algorithms will be discussed in Section 4.

3.4. Final processing

The semantic analysis and the detection report units are part of the final processing. The results after the intermediate processing of the SPD framework show some false positives. In the

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
A	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
B	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
C	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
H	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
I	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0
K	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
N	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
O	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

Legends	
com.mekya.AddFriend	A
com.mekya.communication.SocketOperator	B
com.mekya.FriendList	C
com.mekya.interfaces.IAppManager	D
com.mekya.Login	E
com.mekya.interfaces.ISocketOperator	F
com.mekya.interfaces.IUpdateData	G
com.mekya.Messaging	H
com.mekya.R	I
com.mekya.services.IMService	J
com.mekya.SignUp	K
com.mekya.tools.FriendController	L
com.mekya.tools.XMLHandler	M
com.mekya.types.FriendInfo	N
com.mekya.types.STATUS	O
com.mekya.UnApprovedFriendList	P

Fig. 5. The filled TSM of SAIM target system with class-to-class relationship information.

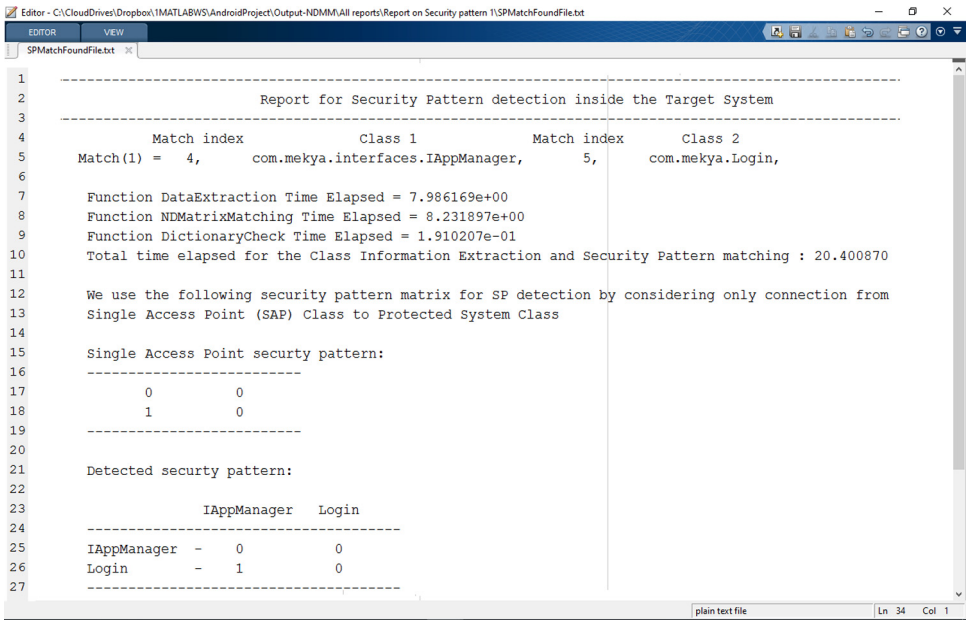

Fig. 6. The report from the SPD tool (time is in seconds).

case of the SAIM target system, four SPMs of the SAP security pattern are found inside the TSM. To find the exact SPM, we apply semantic analysis. The semantic analysis using the dictionary data comparison process reduces false positives to one SPM, which is the actual SAP security pattern. The detection report for the SAIM target system is shown in Fig. 6. The output of the SPD framework provides information on the existence or non-existence and the location of a security pattern.

4. Security pattern matching techniques

Security pattern matching unit is part of the SPD framework, as shown in Fig. 3. The matching problem is well-known in the field of algorithm design as the subgraph isomorphism problem (Lee et al., 2012). At first, for matrix matching in the SPD framework, we use the subgraph isomorphism technique, called the ordered matrix matching (OMM) technique (Alvi and Zulkernine, 2017). However, after finding the challenges of utilizing the OMM technique with the security pattern detection framework, we develop a state-of-the-art matching technique called non-uniform distributed matrix matching (NDMM). For the sake of simplicity, we develop an example to explain the matching techniques. Consider the Single Access Point (SAP) security pattern that implements the School Semester Scheduling (SSS) system. The SAP security pattern, the SSS target system's UML diagrams, their graph and matrix representations are shown in Figs. 4, 7, 8, 9, and 10, respectively.

Now we explain the SAP security pattern and how to extract the security pattern graph using a simple example. We use a class diagram of the SAP security pattern from the security pattern catalog (Schumacher et al., 2006) as shown in Fig. 4(a). The purpose of the SAP security pattern is to allow every legitimate user to gain access to the target system and stop an attacker from unauthorized access. The Windows operating system login screen is an example of the SAP security pattern.

The class diagram is the structural view of the SAP pattern and consists of four classes. We analyze the SAP security pattern, including its classes and their associations, and the interaction of their objects in a usual scenario, as shown in Fig. 4(b). The Client

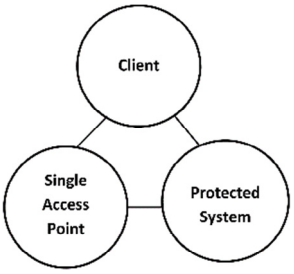

Fig. 7. The SAP security pattern graph.

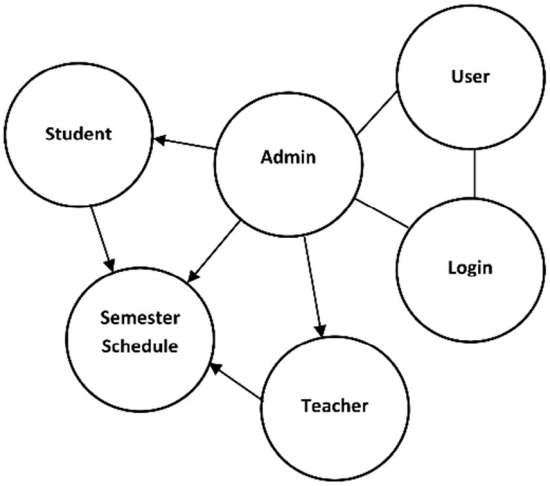

Fig. 9. The SSS target system graph.

class may be a system or an actor (human), while the Boundary Protection class represents a virtual boundary for the client to access resources on the target system. In a static structure, the Boundary Protection class has no physical existence (Bunke and Sohr, 2011; Schumacher et al., 2006). We consider the Client class as a class that serves to keep client information and is used to access the target system. The Protected System class is the representation of the rest of the system (including all classes), considered as a resource. The SAP security pattern's class is utilized for checking the credentials of a user who wishes to enter and access resources. All the associations between classes are shown as two-way associations.

The sequence diagram elaborates more specific dynamic behavior of the given SAP scenario, as shown in Fig. 4(b). A sequence diagram provides a clear and unique view of the dependencies among classes. The sequence diagram shows the interaction between the Client and the SAP classes. The Client class may request permission from the SAP class to access the Protected System class. After checking the user ID/password, the SAP class allows the Client class to access the Protected System class. The decision is made after the verification of the client's credentials. Therefore, the communication from the SAP class to the Protected System class exists. If the permission from the SAP class is granted, then the Client class may access the Protected System class, which contains the resources of the target system. Therefore, through the sequence diagram, we verify the messages between class objects. We represent the communications between security pattern classes in the security pattern graph, as shown in Fig. 7.

We develop the School Semester Scheduling system (SSS) as an experimental subject. It has six classes: User, Login, Admin, Student, Semester Schedule, and Teacher. The relationship among classes is either bidirectional or directed association, as shown in Fig. 8. We implement the exact Single Access Point security pattern as shown in Fig. 4 by considering the Client class as a class for keeping client information (Class 'User' in SSS target systems is the class 'Client' in the SAP security pattern). The detailed knowledge of the SSS target system, such as types of class relationships and cardinality are employed to provide a clear picture of the actual security pattern and to remove ambiguity.

The class User's object wants to access the classes Student, Semester Schedule, and Teacher. However, access to these system resources can be granted after the verification through the classes

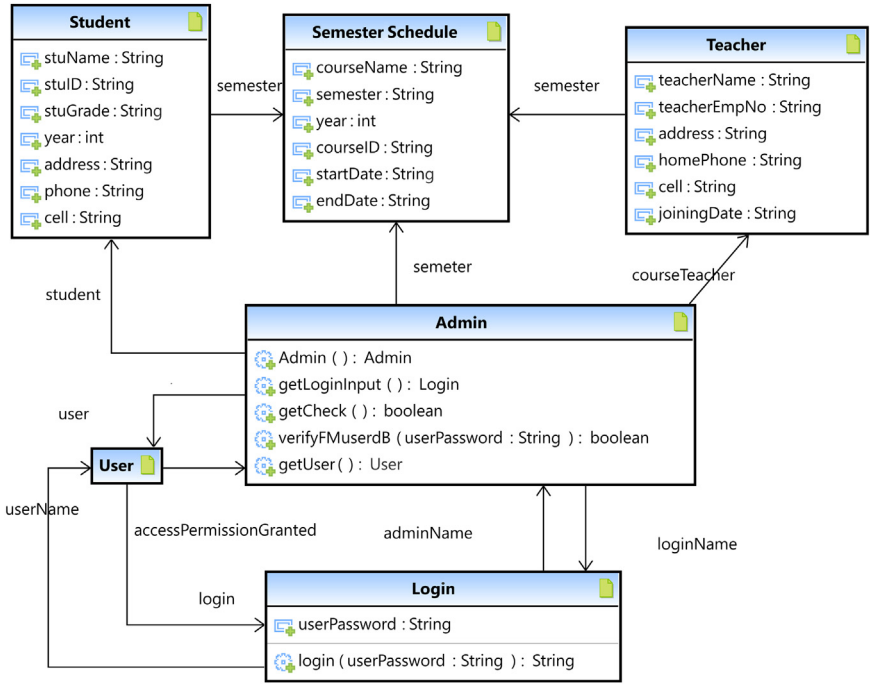

Fig. 8. The SSS target system class diagram.

(a)

	A	B	P	Notation	Classes
A	0	1	1	A	Client
B	1	0	1	B	Single Access Point
P	1	1	0	P	Protected System

(b)

	A	B	C	D	E	F	Notation	Classes
A	0	1	1	0	0	0	A	User
B	1	0	1	0	0	0	B	Login
C	1	1	0	1	1	1	C	Admin
D	0	0	0	0	0	0	D	Semester Schedule
E	0	0	0	1	0	0	E	Student
F	0	0	0	1	0	0	F	Teacher

Fig. 10. Matrix representation of (a) the SAP security pattern and (b) the SSS target system.

Login and Admin. The class Login takes credentials of the object of the class User. The verification is performed through the class Admin. It is a common scenario that all types of user's credentials are stored in a database, and the class Admin accesses the database for verification purposes. After the confirmation of the user's legitimacy, access to the resources (i.e., the classes Student, Semester Schedule, and Teacher) is granted. The class Login plays a role as a Single Access Point security pattern.

We transform UML class diagrams of the SAP security pattern and the SSS target system into graphs. The security pattern graph is available as a class diagram in the security pattern documentation. The interaction between classes is elaborated using a sequence diagram. Every class is represented as a node in the graph. The relationship of a class to class is designated as the vertex of a graph. The process of transformation from a class diagram to a security pattern graph is straightforward. Also, for the whole security pattern catalog, we have a finite number of 'security patterns'. Therefore, we manually develop all security pattern graphs and feed them to the graph database as a matrix entity. The SAP security pattern's UML and graph diagrams are shown in Figs. 4 and 7, and the SSS target system's UML and graph diagrams are shown in Figs. 8 and 9, respectively. The matrix transformation of the SAP security pattern from Fig. 7 and the target software system from Fig. 9 are shown in Fig. 10. It is observed that the classes have the same order in both the first row and the first column of the matrices, where '1' represents the existence of a connection between classes and '0' means no connection.

4.1. Ordered matrix matching technique

The ordered matrix matching (OMM) technique is developed for the detection of an ordered submatrix inside a larger matrix. The detection of a submatrix in a larger matrix is similar to the problem that detects a subgraph inside a larger graph is known as a subgraph isomorphism problem. The subgraph isomorphism problem is NP-complete. However, some subgraph isomorphism algorithms show polynomial-time execution (Fig, 2016). Many efficient subgraph isomorphism algorithms have been proposed in recent years, for instance, GraphQL, SPath, and QuickSI (Lee et al., 2012).

We develop an algorithm as a variant of the subgraph isomorphism problem called the OMM algorithm that executes in polynomial time (Alvi and Zulkernine, 2017; Konagaya et al., 2014). The pseudocode is shown in Algorithm 1. The ordered matrix matching provides the exact submatrix inside the target system matrix. We know that the *n!* class permutations are possible for the arrangement of target system classes. A specific order of classes is used to develop each arrangement and a unique target system matrix (TSM). The security pattern matrix (SPM) may have *m* variants based on the extraction procedure discussed earlier in the beginning of Section 4. Therefore, if we have *m* variants of SPM and *n!* class arrangements of the TSM, then in the worst case, the detection process must run $m \times n!$ times to detect the security pattern inside the target system (if it exists). It is possible that by chance, the SPM may be detected earlier in an exhaustive search.

The possibility to identify a security pattern inside a target system using the OMM technique requires selecting one SPM and starting to search inside the chosen TSM. The searching process moves to another TSM until all *n!* TSMs are checked. Then the SPD tool selects another variant of the SPM and starts the same procedure. It is observed that the exact SPM is detected using an exclusive arrangement of a class permutation of the TSM. However, the detected SPM is not necessarily the actual SPM. The process is highly time and memory consuming and is not favorable to use for a large system.

4.1.1. OMM example

Consider Fig. 10 as an example to explain Algorithm 1. After checking the given conditions with *X* and *Y* matrices, i.e., *X* and *Y* matrices have numeric entries and matrix *Y* must be smaller than matrix *X*. The OMM algorithm converts matrix *X* into a single row vector using a transpose function. The result is shown below:

Row vector of X = [0 1 1 0 0 0 1 0 1 0 0 0 1 1 0 0 0 0 0 0 1 0 1
1 0 0 1 0 0 0 0 0 1 0 00]

It finds the occurrences of the first column of *Y* inside the row vector of *X* using strfind() function in MATLAB:
First column of Y = [0 1 1]

Row vector of X = [**0 1 1** 0 0 0 1 0 1 0 0 **0 1 1** 0 0 0 0 0 0 1 **0 1**
1 0 0 1 0 0 0 0 0 1 0 0 0]

Occurrences = [1 12 22] //shown as bold in Row vector of *X*

The bold elements show the matching with the first column of matrix *Y* three times. However, after eliminating wrap around in step 4 of the OMM algorithm, the detected first column of *Y* inside *X* is reduced to two matches, as shown below.

Row vector of X = [**0 1 1** 0 0 0 1 0 1 0 0 0 1 1 0 0 0 0 0 0 1 **0**
1 1 0 0 1 0 0 0 0 0 1 0 0 0]

idx1 = [1 22]

After the first attempt using while loop with the condition (∼isempty (idx1) && count<= #column in *Y*), the occurrences of the second column of *Y* in *X* can be found as follows:

Second column of Y = [1 0 1]

Row vector of X = [0 1 1 0 0 0 **1 0 1** 0 0 0 1 1 0 0 0 0 0 0 **1 0 1**
1 0 0 1 0 0 0 0 0 1 0 0 0]

Occurrences = [7 21] //shown as bold in Row vector of *X*

idx1 = 1; Calculate the index of the correct occurrence
(match with the previous value).

The OMM algorithm provides the occurrences of the third column of *Y* in *X* as follows:

Third column of Y = [1 1 0]

Row vector of X = [0 **1 1 0** 0 0 1 0 1 0 0 0 **1 1 0**
0 0 0 0 0 1 0 **1 1 0** 0 1 0 0 0 0 0 1 0 0 0]

Algorithm 1: Ordered matrix matching.
Input: Matrix *X* and *Y*, two-dimensional square matrices, where dim(*Y*) < dim(*X*) and element(*Y*) ⊂ *X*.
Output: Location indexes of all the first elements of the matched matrices.

Procedure: OMM (*X*, *Y*)
/* *X* = Target System matrix (TSM) & *Y* = Security pattern matrix (SPM)*/

1. Check inputs:
   a. *X* and *Y* matrices must have numeric entries.
   b. There are only two matrices as inputs, i.e., *X* and *Y* matrices, provided that size(*X*) > size(*Y*).
2. Convert matrix *X* into single row vectors using the TRANSPOSE function.
3. Use the FIND function to detect the occurrences of the first column of matrix *Y* inside a matrix *X* (Get the position of the first element of matrix *Y* of the first column inside matrix *X*).
4. Check and eliminate wrap-around in the given index' s row vector of the occurrences (idx1) inside matrix *X* from step 3.
5. Use WHILE loop with condition ((idx1 != empty) && (count <= #columns in *Y*)), where count = 2.
   a. Use the FIND function to detect the successive columns using the current column of matrix *Y* that may represent matrix *Y* inside matrix *X* (all columns of matrix *Y* will be searched one by one).
   b. Match previous values of row index vector (idx1) of successive columns of matrix *Y* inside matrix *X* with the successive columns of matrix *Y* using the current column of matrix *Y*.
   c. Increment while loop counter (count).
6. Compute the second coordinate index vector (idx2) for successive columns of matrix *Y* inside matrix *X*.
7. The pair of coordinate vectors (idx1, idx2) represents the locations of the first elements of all matrices *Y* inside matrix *X*.
8. Select all submatrices *Y* whose coordinates lie on the diagonal of the matrix *X* in a given sequence.

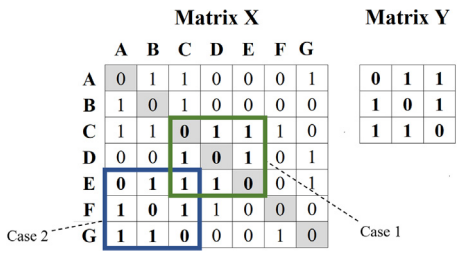

Fig. 11. The diagonal of submatrix *Y* lies inside the diagonal of the matrix *X*.

Occurrences = [2 13 23] (as shown in bold in the above row vector of *X*)
idx1 = 1; Calculate the index of the correct occurrence (match with the previous value).

Finally, compute the other coordinate for the first element of matrix *Y*, which is idx2=1. Therefore, the indices of the location of matrix *Y* inside matrix *X* is (idx1, idx2) = (1, 1)

In summary, the algorithm selects and searches columns from matrix *Y* one by one inside matrix *X* using the strfind() function. The result of a column's locations on every successive increment for the selected column of matrix *Y* inside matrix *X* is matched with the previous result. Finally, the result is the two vectors of linear indexes representing the coordinates of the first element of matrix *Y* inside matrix *X* if more than one matrix *Y* is available inside matrix *X*. We only consider all detected *Y* matrices inside matrix *X* if their diagonals have coincided with the diagonal of matrix *X*. We consider only those detected *Y* matrices inside matrix *X* if their diagonals have coincided. The reason for this consideration is explained in the next paragraphs.

In the OMM algorithm, we only consider a list of detected coordinates of matrix *Y* that are represented as the diagonal elements of matrix *X*. It is explained in Fig. 11, where the list of rows and columns are represented by the same sequence of classes of the matrix *X*. Therefore, the same list of classes and their interactions only can occur in the diagonal of matrix *X* in a given sequence. By considering both submatrices inside *X* shown in Fig. 11, we observe that corresponding classes are as follows:
Case 1: Submatrix *Y* with green border = Classes are C, D, E corresponding to Classes C, D, E
Case 2: Submatrix *Y* with blue border = Classes are A, B, C corresponding to Classes E, F, G

In Case 1, the set of classes are the same in a row and a column, and it is only possible if the diagonal of submatrix *Y* coincides with the diagonal of matrix *X*. However, in Case 2, the set of classes is not the same, and it is only possible if the diagonal of submatrix *Y* does not coincide with the diagonal of matrix *X*. Algorithm 1 is developed for only Case 1. The reason is that the submatrix is the security pattern matrix (SPM), and for all sets of classes, row and column must be the same.

In the next section, we propose a new algorithm for the matching problem. The algorithm works on the class-to-class relationships (C2C) for the detection process. It removes the shortcomings of the OMM algorithm and provides more flexibility and scalability for the detection of security patterns.

4.2. Non-uniform distributed matrix matching technique

The non-uniform distributed matrix matching (NDMM) problem is unique in pattern matching. This problem is partially a subgraph isomorphism problem. The problem can be defined as follows:

In a computational problem, two relation matrices X and Y are given as input (provided that the size of matrix X is greater than the size of matrix Y). It is required to determine whether matrix X contains an ordered matrix Y as a submatrix or if the matrix Y elements are distributed non-uniformly inside matrix X.

The mathematical representation of the target system class diagram is the TSM. It reveals that the order of the class names will change the entire matrix into another matrix arrangement. The transformation of the TSM matrix from Figs. 10 to 12 reveals that classes have one-to-one relationships. Therefore, the detection of the SPM inside the TSM is the class-to-class matching problem (since, in some cases, class order in the TSM may transform the problem into the actual submatrix isomorphic problem).

The advantage of the algorithm is that it needs only one TSM to detect the implemented security pattern. Fig. 12 shows that the SPM elements are scattered inside the TSM. Therefore, in the selection of TSM out of *n!* TSM's, it is highly unlikely that the selected TSM has SPM, which is visible as a submatrix. Therefore, the class-to-class matching problem is dominant over the submatrix isomorphic problem in the matching process of the SPM inside the TSM.

Algorithm 2: Non-Uniform Distributed Matrix Matching.
Input: Matrix *X* and *Y*, two-dimensional square matrices, where dim(*Y*) < dim(*X*) and element(*Y*) ⊂ X.
Output: The result is location indices of all the elements of the matched SP matrices.

Procedure: NDMatrixMatching (*X*, *Y*)
/* *X* = Target system matrix (TSM) & *Y* = Security pattern matrix (SPM)*/

```
1-  LOAD X (x, y) and Y (m, n) matrices
2-      INITIALIZE MatchList
3-      M ← MaxNumOfXCol - (MaxNumOfYCol - 1)          /*Possible comparisons*/
4-      while i <= MaxNumOfXRow                          /*Row comparison*/
5-        while a <= M
6-          if Y (YRowIndex, YColIndex) =  X (XRowIndex,  XColIndex)
7-            MatchList ← [XRowIndex,  XColIndex]
8-            while b <= M + 1
9-              if Y (YRowIndex, YColIndex) =  X ( XRowIndex,  XColIndex)
10-               MatchList ← [XRowIndex,  XColIndex]
11-               while c <= M + 2
12-                 if Y (YRowIndex, YColIndex) =  X (XRowIndex,  XColIndex)
13-                   MatchList ← [ XRowIndex,  XColIndex]
14-                   ExtractedPattern ← EXTRACT (MatchList)
15-                 if Y = ExtractedPattern
16-                   Matched ← STORE MatchList      /*List of matched classes*/
17-                 end if
18-               end if
19-             end while
20-           end if
21-         end while
22-       end if
23-     end while
24-   end while
```

(a)

	A	B	P
A	**0**	**1**	**1**
B	**1**	**0**	**1**
P	**1**	**1**	**0**

Notation	Classes
A	Client
B	Single Access Point
P	Protected System

(b)

	A	D	B	E	C	F
A	**0**	0	**1**	0	**1**	0
D	0	0	0	0	0	0
B	**1**	0	**0**	0	**1**	0
E	0	1	0	0	0	0
C	**1**	1	**1**	1	**0**	1
F	0	1	0	0	0	0

Notation	Classes
A	User
B	Login
C	Admin
D	Semester Schedule
E	Student
F	Teacher

Fig. 12. Matrix representation of (a) SAP security pattern and (b) another SSS target system.

The SPM is detected inside the TSM by the detection of the same class relationship of the security pattern classes among the target system classes. Therefore, some of the target system classes may form security pattern class relationships as a non-uniformly distributed form. However, a one-to-one relationship exists in any case of the target system class arrangements. We exploit this understanding and develop a state-of-the-art algorithm to detect a security pattern inside a target system without considering the order of the target system classes. The C2C matching algorithm pseudocode is shown in Algorithm 2.

4.2.1. NDMM example

We consider a running example for illustrating the NDMM algorithm using *X* and *Y* matrices from Fig. 12.

Step 1: Find the first-row elements of *Y* inside the first row of matrix *X*.

The first row of matrix *Y* (0 1 1) is selected by the SPD tool, which then starts comparing the first element '0' with the elements in the first row of matrix *X* (0 0 1 0 1 0) until it finds a match. The first element of matrix *Y* (i.e., '0') is the same as the first element in the first row of matrix *X* (i.e., '0'). Therefore, class A is selected. The search starts with another element in the remaining elements of the first row of matrix *X* until it is found in the column of class B (i.e., '1'). Then the SPD tool starts to find the last element '1'of matrix *Y* in the remaining elements of the first row of matrix *X*. The third element is found in the column class C. Therefore, we have found the first row of matrix *Y* inside the elements of the first row of matrix *X*. The first matched list of classes and their positions in matrix *X* are 1, 3, and 5 (i.e., classes A, B, and C). This match stores into *MatchList* array.

Step 2: Make an extracted matrix based on the matching found in step 1.

The match positions 1, 3, and 5 show that class A of matrix *X* has relationships with the classes A, B, and C of the first row of matrix *X* which is similar to class A of matrix *Y* that has relationships with classes A, B, and P of the first row of the SPM. Therefore, there may be a chance that the relationships from class B to classes A, B, and C, and from class C to classes A, B, and C will be similar to those in the SPM. Now, we extract the relationships of class B in matrix *X* from classes A, B, and C, and class C of matrix *X* from classes A, B, and C. Then we form the extracted pattern matrix (EPM) and compare it with the matrix *Y*. If both are equal, then we have found the first matrix *Y* scattered as a non-uniformly distributed matrix inside matrix *X*.

Step 3: Repeat step 1 and step 2 for all the elements of matrix *X*

After successful or unsuccessful matching, the comparison of the first row of the matrix Y with the first row of the matrix X will continue. The first element of the first row of matrix Y is compared with the second element of the first row of the matrix

X, and the comparison continues in this way. When both are '0', we start to compare the second element of matrix Y to the third element of matrix X. In the end, the second match is found with classes D, B, and C with positions 2, 3, and 5 in matrix X. This process forms the EPM and compares it with matrix Y. If the result is successful, it stores the EPM class positions from matrix X into the *MatchList* array. The outermost while loop continues to scan the whole matrix X in this way and stores all the lists of the three classes (used to form matrix Y) into *MatchList* array.

4.3. Development of the multi-level security pattern dictionary using semantic analysis

In this paper, we use semantic analysis to develop a multi-level security pattern dictionary (MSPD). The purpose of using a multi-level dictionary is to verify the actual security pattern matrix among all extracted matrices using Algorithm 1 and 2. We discuss the development of MSPD level by level, and the data collected at each level is unique. The construction of MSPD from the first level to the fourth level is described as follows.

First level: In the first level, the dictionary is prepared using the selection of its data elements from the security pattern document. In the security pattern documentation, these words are available under the heading 'Also Known As'. For example, in the case of Single Access Point (SAP) security pattern, the following are the available elements under the heading 'Also Known As' given in Schumacher's catalog (Schumacher et al., 2006) as follows.

One Way In, Login Window, Guard Door, or Validation Screen

In the first level of MSPD, we use the single words from the list of words given under the heading of 'Also Known As' from a security pattern document. We also combined them as camel case (Binkley et al., 2009). In the camel case, we attach the first and the second-word keeping upper case letters and form a single word, such as OneWay. We change the syntax of the phrase and generate variations using new words based on different programming practices by developers, such as a combination of lower and uppercases, use of '_' and '–' to connect words. Then, we can generate many variations (Jones, 2019; Rossum et al., 2013). The generated words are shown below:

Simple list: OneWay, Login, Window, Guard, Door, Validation, Screen

Created a list of variations: Oneway, Login_Window, Guard_Door, Validation_Screen

Second level: In the second level, we select synonyms of available words at the first level of MSPD, excluding the words themselves and make other possible names as follows:

Login	Register, Enter, Log on
Window	Screen, Interface, Bay, Aperture
Guard	Secure, Protect, Shield
Door	Port, Socket, Porta, Entrance
Validation	Acceptance, Authorization, Proof, Verification
Screen	Display, Monitor, Window

Further, we introduce many words by intuition that developers may use as class names in a security pattern. It is recommended that the security pattern community participate in promoting and incorporating more reasonable security pattern names or aliases under the template element, 'Also Known As' in the security pattern documentation. Therefore, in this way, the reliability of semantic analysis for the detection process will be increased. The example of a list of words suggested by developers based on intuition is as follows.

Suggested by developers: Logindialog, LoginSettingsDialog, Loginscreen, Signon

Third level: This level of the dictionary is developed using a Bag-of-Word model, also known as the Unigram model of natural language processing (NLP). It can be enhanced to the N-gram model of NLP for developing the fourth-level of the security pattern dictionary. We have considered two documents: a security pattern document (SPD) and a software requirement document (SRD). We have chosen SRD because many class names come from it. For example, in the case of the ATM system's SRD, a software designer takes the statement "David withdraws CAD-300.00/- from his account", and extracts classes and its attributes. The words "Customer" and "Account" are viewed as classes, and "Withdraw" is conceived as a method of class Account, where CAD-300.00/- is used to create a floating-point variable of the class Account. Therefore, the above example provides us with the importance of software requirement document (SRD) to extract nouns and verbs that may be used as class names and methods, respectively.

Fig. 13 shows the way of parsing two documents, one is the SPD, and the other is SRD. By including SRD, we extend the list of possible words that may be used as class names. Using these documents, e.g., SAP documents (from Schumacher's catalog) and software requirement document of the subject software, the dictionary development process filters all nouns and verbs from the text using the Unigram model.

Fourth level: The fourth level is the enhanced version of the third level of MSPD. We use the N-gram model of NLP for developing the fourth-level of the security pattern dictionary, where $N = 2$. Therefore, it is also known as the Bigram model of NLP. We implemented the Bigram model in MATLAB R2018a. The program uses the training data with the use of tagging for the parts of speech (POS) of the sentence. For example, a proper noun is tagged as PNN, a noun is tagged as NN, the pronoun is tagged as PN, the verb is tagged as VB, the adjective is tagged as ADV, and the preposition is tagged as PP. The given sentences are parsed based on POS. The accuracy in the result is increased with the increased information in training data. The detailed working of the algorithm is shown in Fig. 13. The final list is the combination of two words (nouns) such as Login Window, Guard Door, or Validation Screen. We use every item from the list that consists of two words with different syntaxes, e.g., LoginWindow, Login_Window, or Login-Window. This process is the standard way to combine words as a class name in different programming languages such as Java and Python.

The semantic analysis checks the class names from the dictionary data with the class names of the detected security pattern matrices (SPMs). The MSPD is used based on levels. The list of words in each level is unique. We utilize all the levels of the list for word searching. If any of the class names match, then there is a high chance that the corresponding SPM represents the actual security pattern. Semantic analysis reduces the number of similar SPMs significantly. Therefore, the probability to determine the actual SPM that describes the exact security pattern is increased with semantic analysis.

5. Experimental evaluation

Security pattern detection effort is in an infant stage. There is no benchmark available for evaluation. However, we use one case study from the work of Bunke and Sohr (2011) for security pattern detection. They use an open-source Android application named Simple Android Instant Messaging Application (Mermerkaya, 2013). Besides, we add three more case studies in our benchmark for the detection of security patterns. These are the School Semester Scheduling system (SSS), Automated Teller Machine simulator (ATM sim.) (Bjork, 2019), and Electronic Voting System (VoteBox) (Sandler et al., 2008). These four case studies use three different kinds of security patterns Single Access Point (SAP), Security Session (SS), and Authenticator (AP) (Schumacher

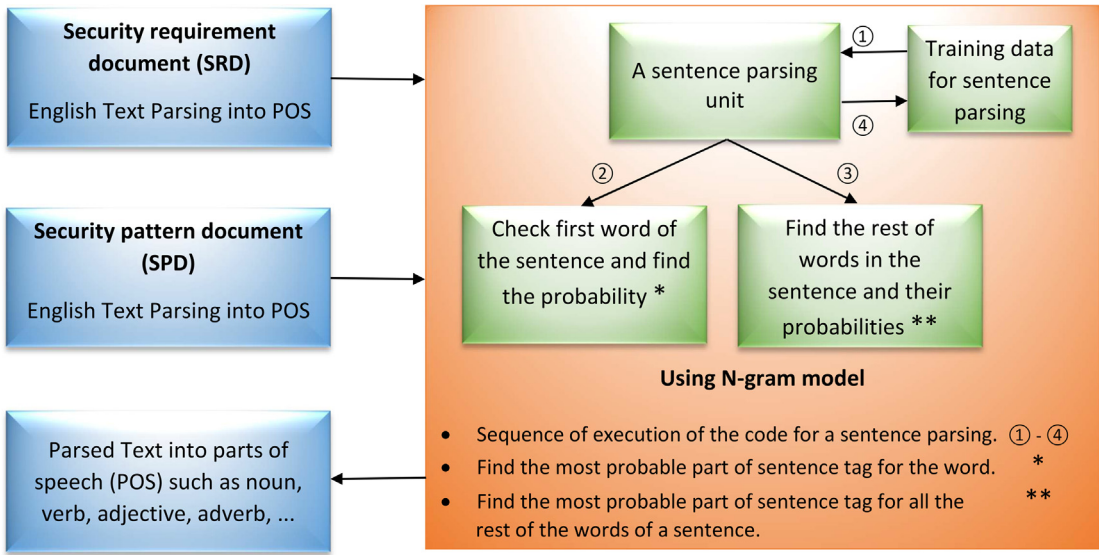

Fig. 13. The parsing of the document using the N-gram model.

et al., 2006). We hope that these case studies may be part of future benchmarks for the detection of security patterns.

The case studies we have chosen are based on the following selection criteria:

(1) use in past research work (i.e., Bunke and Sohr (2011)),
(2) present diversity in the set of the problem domain of case studies,
(3) provide a diversity of the use of security patterns,
(4) have different sizes of the source code based on the number of classes,
(5) exist an appropriate number of relationships among classes,
(6) increase the complexity of the case studies based on class to class relationships,
(7) selected from open-source software projects, and
(8) developed in Java programming language.

The following evaluation metrics are used in the experiments:

(1) Performance of the detection technique
  (a) accuracy of the detection of security pattern
  (b) computational time of the whole process
  (c) computational time of the matching and semantic analysis
  (d) location of security pattern the source code of the given case study

(2) Effect of complexity on performance
  (a) size of source code, number of classes, and relationships among the classes of a target system and security pattern
  (b) arrangement of the elements of the target system matrix
  (c) different types of security patterns

(3) Relationship between complexity and memory utilization
  (a) impact of the number of classes in target systems on memory utilization
  (b) impact of relationships between classes in target systems on memory utilization

We observe that the selected target systems lack a standard implementation of security patterns. It is challenging to find target systems that have used security patterns properly. To address this difficulty, we implement the Authenticator pattern only in VoteBox for the detection experiments. We detect three security patterns, two from System Access Control Architecture, namely, Single Access Point (SAP) and Security Session (SS), and one from Operating System Access Control, i.e., Authenticator (AP). The documentation of these security patterns is available in a catalog developed by Schumacher et al. (2006). We effectuate the security pattern detection framework, including matching algorithms employing the SPD tool. The tool is developed in MATLAB R2018a software and used for the detection of security patterns from the target systems. We use a computer with Intel Core 2 Duo Processor P7350, including 4 GB of RAM, 3 MB Cache, processor base frequency 2.00 GHz and FSB speed 1066 MHz for experiments.

Table 1 provides the information about the target systems and security patterns that are used for the detection process. The SAP security pattern is employed with different sizes of the SPMs; for example, in the SSS project, SAP pattern matrix size is 3 x 3, and in the SAIM project, SAP pattern matrix size is 2 x 2. The size of the SAP depends on the inclusion or exclusion of the Client class. The SAP security pattern's graph and matrix are shown in Figs. 7 and 10. In the SSS project, the client is considered as part of a software system and used as a Java class. However, in the SAIM project, a client is a human being, not a Java class. Therefore, the Client class is eliminated, and the size of the SPM becomes 2 x 2.

We evaluate the overall time consumption and the fundamental SPD processes, i.e., data extraction, matrix matching, and semantic analysis while utilizing the OMM and NDMM matrix matching algorithms separately.

5.1. Detection results

5.1.1. OMM experiments

The detection results for the security pattern detection tool using the OMM technique are shown in Table 2. The table shows that when we run the SPD tool for all the class permutations of TSMs (i.e., *n!* TSMs) one-by-one, then we find the SPM in one of the TSMs. However, we randomly select a TSM from *n!* TSMs and apply the SPD tool. We observe that with a random selection of a TSM, it is not necessary to find a security pattern from the selected TSM.

During multiple time executions over *n!* times, the SPD tool may report some false positives. The semantic analysis process

Table 1
Size and complexity of TSM and SPM.

Case studies (Target systems)	Size and complexity of TSM			Security pattern	Size and complexity of SPM		
	No. of classes	No. of class interactions	Size of TSM		No. of classes	No. of class interactions	Size of SPM
1 - SSS	6	11	6 x 6	SAP	3	6	3 x 3
2 - SAIM	16	14	16 x 16	SAP	2	1	2 x 2
3 - ATM sim.	38	24	38 x 38	SS	5	8	3 x 3
4 - VoteBox	137	170	137 x 137	AP	3	4	3 x 3

Legends:

SSS	School Semester Scheduling system	SAP	Single Access Point
SAIM	Simple Android Instant Messaging	SS	Security Session
ATM sim.	ATM system simulation	AP	Authenticator
VoteBox	Electronic voting system		

participates in filtering the actual security pattern from false positives. After the detection of the actual security pattern, the location of the security pattern matrix with corresponding class names inside a target system is detected. The location of the security pattern may be used for tracking security patterns during the runtime of the software system to verify its implementation standards, measure security quality, test for expected security loopholes, and to monitor its life cycle.

Fig. 14 shows the computational time for major processes using the SPD tool. The computational time is measured entirely for one execution for detecting a security pattern matrix (SPM) inside that target system matrix (TSM). The graph shows that the computational time for the whole SPD tool rises with the increasing size and complexity of the target system. The major contributor to the execution time is the data extraction process. The computational time of the data extraction process contributes only once in the execution of the detection process. However, the SPDT tool executes the matching process multiple times for *n!* TSMs. The computational time for the ordered matrix matching (OMM) and semantic analysis processes is significantly low, i.e., the OMM process takes 0.0010 s to 0.1122 s, and the semantic analysis process takes from 0.0017 s to 0.0027 s for the target systems 1 to 4, respectively. Besides, the computational time of the matching process increases significantly in ATM sim. target system.

The selection of a TSM from *n!* arrangements of the TSM has a significant effect on computational time. We evaluated this fact in Section 5.2.1 using five different TSMs of School Semester Scheduling Systems and measure the elapsed time for the detection process. Therefore, a choice of TSMs is critical because every TSM has a unique arrangement of elements. However, in Table 2, the results shown in the 3rd column are randomly selected arrangements of TSMs from the set of *n!* arrangements of TSMs. The detection result based on a random selection of TSMs is not substantial; whereas, the result shown in the 4th column is based on an exhausted detection process by running the SPD tool on each of the *n!* arrangements of every TSM. The SPD tool positively detects the SPM. However, the computational time is very high because the detection process runs using every *n!* TSMs or stops until it finds the SPM. The execution time for the SPD tool may vary based on the detection of the SPM; it may stop at an earlier part of the list of TSMs, or in the worst-case scenario, at the last TSM. Additionally, the semantic analysis process is in place to remove false positives; therefore, we have zero false positives.

5.1.2. NDMM experiments

The NDMM algorithm works based on class-to-class relationship matching; therefore, any one of the TSMs from the set of *n!* TSMs are enough for SPM detection. The detection results for the security pattern using the NDMM technique are shown in Table 3.

The NDMM technique matches all three target systems (i.e., SSS, ATM sim. and VoteBox from Table 3) perfectly; however, in the case of the SAIM target system, its output is 35 matches which reduces to 1 after semantic analysis. Why 35 matches found in the case of the SAIM case study for SAP 2 x 2 matrix pattern? The possible reason is the size and class interactions in the SPM of the SAP security pattern. Table 1 shows that the classes of the SAP security pattern have only one relationship. It means that only one element has a value of '1' out of 4 elements of 2 x 2 size of the SPM of the SAP security pattern, and all other elements are 0. On the other side, the target system has a small number of relationships with the available positions in the TSM. The number of cells inside the TSM is 16 x 16 = 256, and out of 256 elements of the TSM, only 14 elements contain '1'. Therefore, it increases the chance of occurrence of the SPM in the TSM. It is observed that smaller size SPM has a higher chance of detecting inside the TSM.

In this case, semantic analysis finds a correct SAP pattern among 35 detected patterns. It analyzes the SAP pattern names given in the SAP documentation and intuitively expected names based on the SAP scenarios. We develop a dictionary including all SAP pattern names of the classes for comparison with the class names extracted from 35 detected patterns. The data in the dictionary is increased, which will help to find the correct SAP pattern. The detailed discussion on the development of a dictionary for semantic analysis is available in Section 4.3. The location of the security pattern is reported by the SPD tool after semantic analysis.

Fig. 15 shows the computational time for the major processes, i.e., data extraction, matrix matching, and semantic analysis. The graph indicates that the computational time for the data extraction rises with the increased number of classes (size) of target systems. The computational time of the matching process increases significantly in ATM sim. target system. Table 1 shows that the SS security pattern is used for detection in ATM sim. and AP security pattern is used for the detection in VotBox. The ratio of the total number of classes with the total number of class interactions for ATM sim. is equal to 24/38 = 0.63, approximately. Similarly, the ratio of the total number of classes with the total number of class interactions for VoteBox is 170/137 = 1.24, where a decrease in ratio means more complexity. It means ATM sim. has complex TSMs compare to VoteBox. Therefore, the only logical reason for a sudden increase in computational time is a unique order and the complexity of interaction relations of class names that are used to form a target system. This phenomenon will be observed in the next section.

Each unique arrangement of classes is from the unique TSM of the target system. The arrangement of the elements inside the TSM is based on a unique order. It has a significant role in the computational time of the process. This fact is evaluated

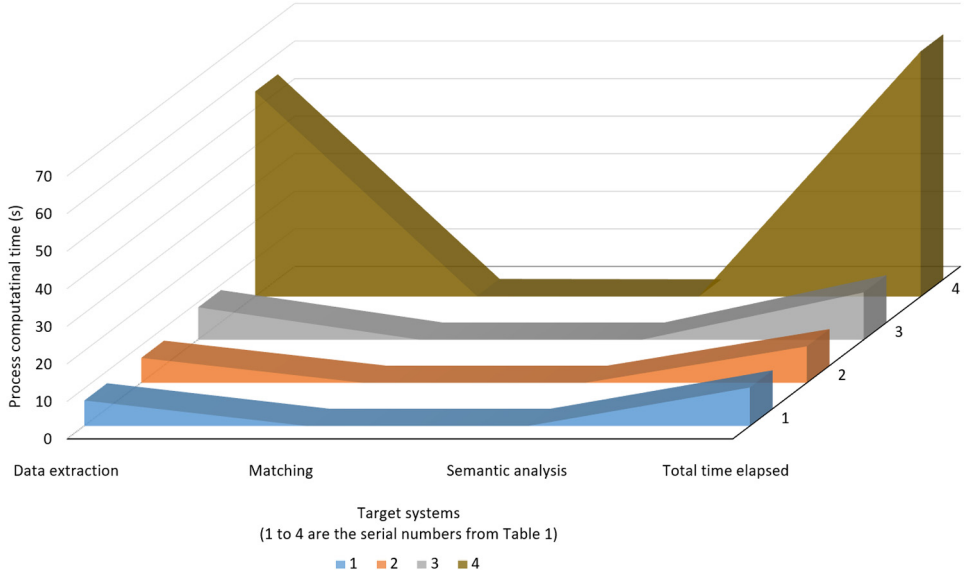

Fig. 14. Performance vs. the SPD processes using the OMM in a successful match.

Table 2
The SPD tool detection results using the OMM algorithm.

Case studies (Target systems)	Security pattern (SP)	Security pattern detected after applying detection tool (detected = 1 or otherwise)		Location of security pattern
		A randomly selected TSM from the set of *n!* target system matrices	For all TSMs in the set of the *n!* target system matrices	
1 - SSS	SAP	0	1	1,2,3
2 - SAIM	SAP	0	1	4,5
3 - ATM sim.	SS	0	1	9,10,11
4 - VoteBox	AP	0	1	5,6,7

Table 3
The SPD tool detection results using the NDMM algorithm.

Case studies (Target systems)	Security pattern (SP)	SP instances detected after matching	SP instances reduces after semantic analysis	Location of Security pattern
1 - SSS	SAP	1	N/A	1,2,3
2 - SAIM	SAP	35	1	4,5
3 - ATM sim.	SS	1	N/A	1,10,13
4 - VoteBox	AP	1	N/A	5,14,137

in Section 5.2 for both the OMM and the NDMM techniques by selecting five TSMs of the same case study. The results are shown in Fig. 18 of Section 5.2.1 for the OMM technique and in Fig. 20 of Section 5.2.2 for the NDMM technique demonstrating the correlation between computation time and unique order or unique arrangement of class names. For the NDMM technique, we randomly select one TSM out of *n!* TSMs. Therefore, the correlation between computation time and unique order or unique arrangement of class names that makes TSM unique is not visible because of the random selection of a single TSM. The semantic analysis process outperforms only in the case of the SAIM target system. In the other target systems, the actual SPM is detected correctly without any variations. Therefore, there are no false positives.

5.2. Effect of complexity on performance

5.2.1. OMM technique

We consider the number of classes, arrangements and their interactions (i.e., relationships) of the target systems as some indicators of complexity that may have effects on the performance of security pattern detection. We calculate the total elapsed time (TET) on four different target systems using the OMM technique. The time elapsed is calculated when security patterns are successfully detected in the target systems. The results are shown in Fig. 16.

Fig. 16(a) reveals that total elapsed time (TET) depends on the number of classes and their interactions with the target systems. We calculate the total elapsed time on four different target systems using the OMM technique. The time elapsed is calculated when security patterns are successfully detected in target systems. Fig. 16(b) reveals that the total elapsed time does not depend on the number of classes and their interactions in the security patterns used in detection.

We randomly selected five arrangements of TSM from the School Semester Scheduling (SSS) system and applied the OMM technique. We want to observe the effect of the change in the arrangements of the TSM on the performance regarding total elapsed time for detection. Fig. 17 shows the five TSM arrangements and Fig. 18 shows the total elapsed time of the corresponding five randomly selected TSMs. The SAP security pattern matrix

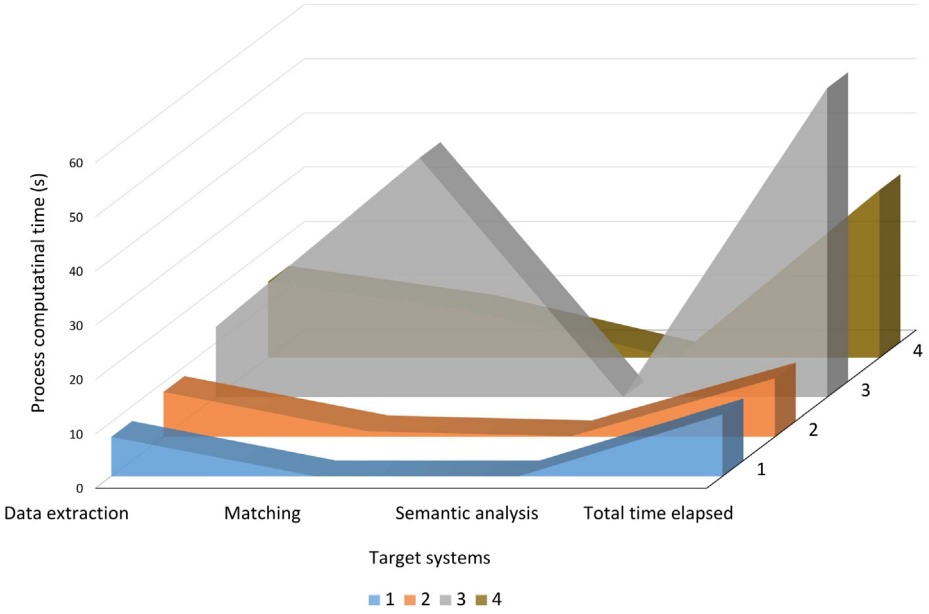

Fig. 15. Performance vs. the SPD processes using the NDMM technique in a successful match.

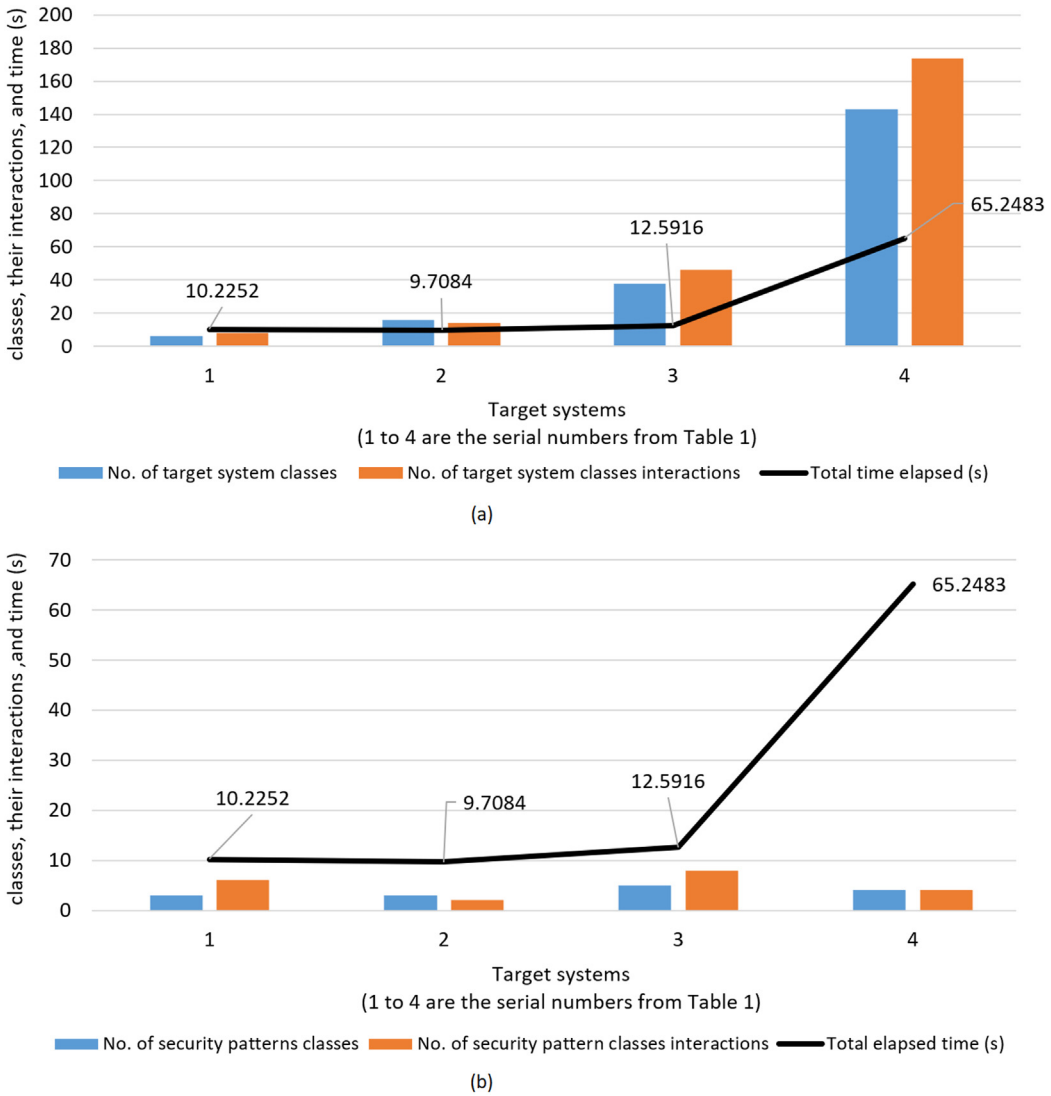

Fig. 16. (a) Target system classes and (b) Security pattern classes, their interactions and total time elapsed (s) in the OMM technique.

TSM1

	A	B	C	D	E	F
A	**0**	**1**	**1**	0	0	0
B	**1**	**0**	**1**	0	0	0
C	**1**	**1**	**0**	1	1	1
D	0	0	0	0	0	0
E	0	0	0	1	0	0
F	0	0	0	1	0	0

TSM2

	F	D	C	B	E	A
F	0	1	0	0	0	0
D	0	0	0	0	0	0
C	1	1	**0**	**1**	1	**1**
B	0	0	**1**	**0**	0	**1**
E	0	1	0	0	0	0
A	0	0	**1**	**1**	0	**0**

TSM3

	C	B	D	E	F	A
C	**0**	**1**	1	1	1	**1**
B	**1**	**0**	0	0	0	**1**
D	0	0	0	0	0	0
E	0	0	1	0	0	0
F	0	0	1	0	0	0
A	**1**	**1**	0	0	0	**0**

TSM4

	A	F	E	D	B	C
A	**0**	0	0	0	**1**	**1**
F	0	0	0	1	0	0
E	0	0	0	1	0	0
D	0	0	0	0	0	0
B	**1**	0	0	0	**0**	**1**
C	**1**	1	1	1	**1**	**0**

TSM5

	E	C	B	F	A	D
E	**0**	0	0	0	**1**	**1**
C	0	0	0	1	0	0
B	0	0	0	1	0	0
F	0	0	0	0	0	0
A	**1**	0	0	0	**0**	**1**
D	**1**	1	1	1	**1**	**0**

Legends

Classes	
User	A
Login	B
Admin	C
Semester Schedule	D
Student	E
Teacher	F

Fig. 17. Different TSM arrangements in the OMM technique.

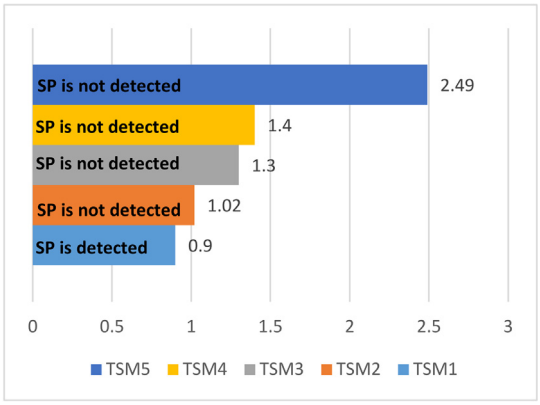

Fig. 18. Different TSMs vs. total time elapsed (s) in the OMM technique.

(SPM) is visible in TSM1; however, with other random selections of TSMs, SPM scatters inside TSM2 to TSM4. The elements of SPM are shown in bold as '0' or '1' inside all TSMs. The SAP security pattern is detected successfully in TSM1 only. Fig. 18 shows that the shortest time is observed in TSM1, where SAP security pattern is detected successfully. But on the other TSMs, the total elapsed time is different. We arrange the TSMs from TSM5 to TSM1, therefore, the graph in Fig. 18 shows decreasing order in total elapsed time (TET) accordingly. This graph indicates that the arrangement of classes inside TSM has a significant effect on the performance.

5.2.2. NDMM technique

We calculate the total elapsed time on four different target systems using the NDMM technique. The time elapsed is calculated when security patterns are successfully detected in the target systems. The results are shown in Fig. 19. Fig. 19(a) reveals that the total elapsed time does not depend on the number of classes and their interactions.

We calculate the total elapsed time on four different target systems using the NDMM technique. The time elapsed is calculated when security patterns are successfully detected in the target systems. Fig. 19(b) reveals that the total elapsed time (TET) does depend on the number of class interactions of security patterns used in the detection. The number of class interactions of security patterns used in target system 3 to target system 4 is decreased with a decrease of total elapsed time; however, the number of classes is increased. On the other hand, from target system 1 to target system 2, the number of classes is the same. Still, the interactions between classes are decreased, and the corresponding total elapsed time is reduced. This reduction shows a correlation between the number of interactions of security pattern classes and the total time elapsed for detection.

We randomly selected five arrangements of TSM from the School Semester Scheduling (SSS) system project, as shown in Fig. 17 and apply the NDMM technique. We have observed the effect of the arrangements of the TSM on the performance regarding total elapsed time (TET). Fig. 20 shows the total elapsed time of the corresponding five randomly selected TSMs. The SAP security pattern matrix (SPM) is visible in TSM1; however, with the other random selections of TSM, SPM scatters inside TSM2 to TSM5. The elements of SPM are shown in bold as '0' or '1' inside all TSMs.

The SAP security pattern is detected successfully in TSM1 to TSM5. Fig. 20 shows that the shortest time is observed in TSM2, where SAP security pattern is detected successfully. However, on the other TSMs, the total elapsed time is different. Fig. 20 shows decreasing order in total elapsed time. This indicates that the arrangement of classes inside TSM has a significant effect on the performance. Every TSM arrangement has a unique impact on the entire elapsed time of the detection process. Therefore, the total elapsed time of each TSM is unique.

Figs. 18 and 20 provide a comparison between the NDMM and the OMM techniques. The OMM technique can detect SPM only from the specific arrangement of TSM out of $n!$ TSMs and the NDMM technique can detect SPM from any arrangement of $n!$ TSMs. Further, the total elapsed time (TET) in case of the NDMM technique (TET @ TSM2 = 0.57 s) is less than that of the OMM technique (TET @ TSM1 = 0.9 s).

5.2.3. Effect of complexity on memory utilization

The complexity of the software projects is going up with the increase in the number of classes, arrangements of classes and their class interactions. The effect on the detection process because of complexity may have an impact on memory utilization and time consumption. In the previous section, we have discussed the implications for time consumption because of the complexity. Table 4 provides the granular level detail of the impact of the complexity of the software system on the utilization of the memory. Fig. 21 presents the effect of complexity on memory utilization during the detection process using the OMM and the NDMM techniques.

Fig. 21(a) and (b) show that an increase in the number of classes and class interactions have a significant influence on the use of memory by the security pattern detection system. However, in the comparison of the OMM and the NDMM techniques, the effect of memory consumption does not differ that much concerning the complexity of the software system. Table 4 shows that the increase in the number of classes and their interactions affect the OMM and the NDMM techniques differently. The NDMM technique shows slightly less use of memory than the OMM technique during detection (in the case of TSM 1, 3, and 4). The purpose of viewing the effect of complexity on memory utilization is useful in large projects. We use open-source java-based projects as case studies that have a limited number of classes. However, large projects that may have thousands of java classes have a visible effect because of complexity on memory utilization. Therefore, the number of classes, relationships between classes, number of methods per class, coupling and other code complexity level metrics have an impact on memory utilization. Using good coding practice, code structure, efficient code writing techniques, and code optimizing tools we can reduce complexity and memory usage for a security pattern detection tool.

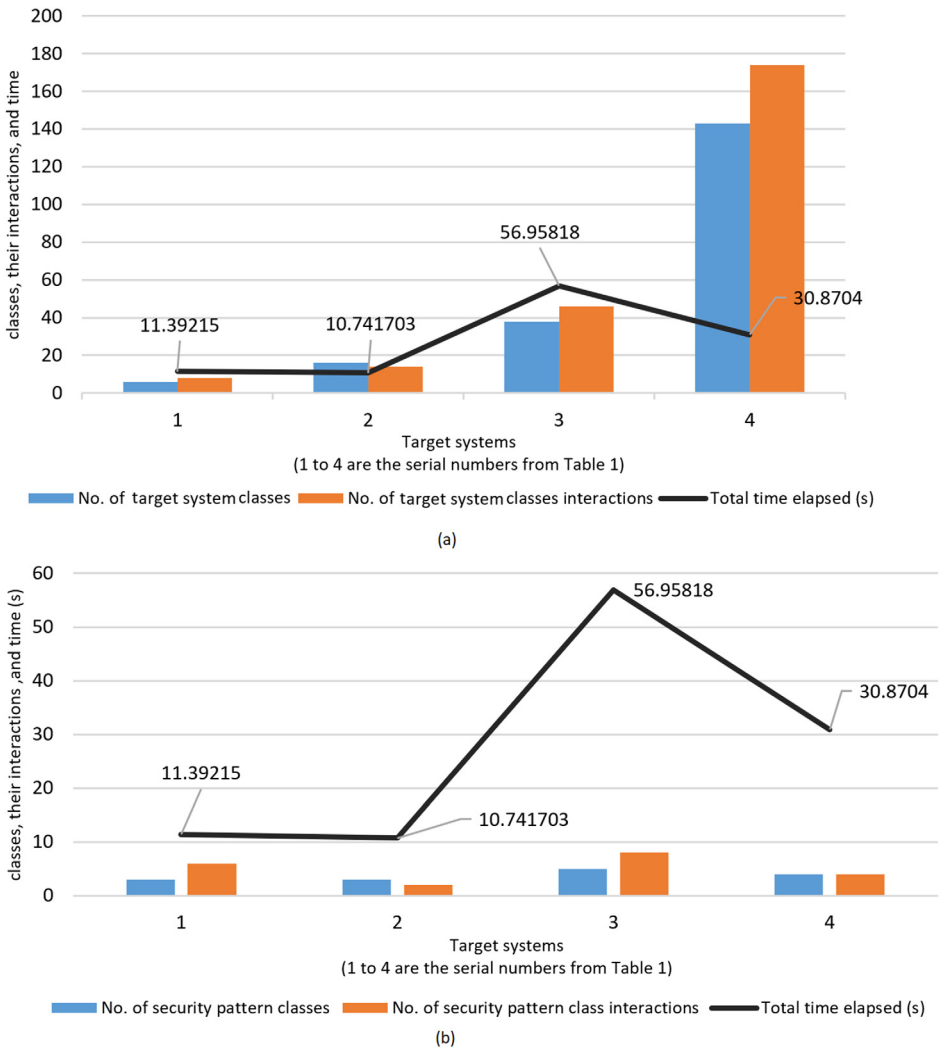

Fig. 19. (a) Target system classes and (b) Security pattern classes, their interactions and total time elapsed (s) for the NDMM technique.

Table 4
Complexity vs. Memory utilization.

Case studies (Target systems)	Security patterns	Complexity		Memory utilization (kB)		Security pattern detected
		No. of classes	No. of class interactions	OMM	NDMM	
1 - SSS	SAP	6	11	8.923828224	8.896484352	Yes
2 - SAIM	SAP	16	14	41.8816	42.7008	Yes
3 - ATM sim.	SS	38	24	199.4140672	199.4023936	Yes
4 - VoteBox	AP	137	170	2634.160128	2634.148864	Yes

6. Related work

Security pattern detection can assure the existence of security patterns and provide the knowledge of their structure and location in the software system source code (Gorton, 2011). The primary research work for the detection of patterns was first carried out in the area of the design patterns. Therefore, a basic guideline exists for pattern detection. Design patterns are not explicitly developed for system security; instead, they discuss the functionality of a software system at the design level. Gamma et al. (1995) produced the first catalog, including 23 design patterns. They classified the catalog based on structural, creational, and behavioral types. On the other hand, Schumacher et al. (2006) produced a catalog with a total of 46 security patterns and categorized them based on security features. However, many security pattern catalogs are available (Fernandez, 2013; Hafiz et al., 2011; Kienzle and Elder, 2019; Kienzle et al., 2019; Slavin and Niu, 2018; Yskout et al., 2006). As an architectural component, security patterns can be placed between design patterns and micro patterns. Security patterns specifically fulfill security requirements, which are non-functional requirements in software engineering processes.

The detection methods for security patterns require more comprehensive details of security features. Security patterns explain the precise steps of security implementation. Furthermore, for solving a security problem, many security patterns may be used together, and they also interact with system design patterns. Therefore, in some cases, the exact solution to a security problem is the system of patterns (Yskout et al., 2006).

The difference in detection is understandable by the definition of design patterns and security patterns and their documentation. The purpose of design patterns is to solve software design

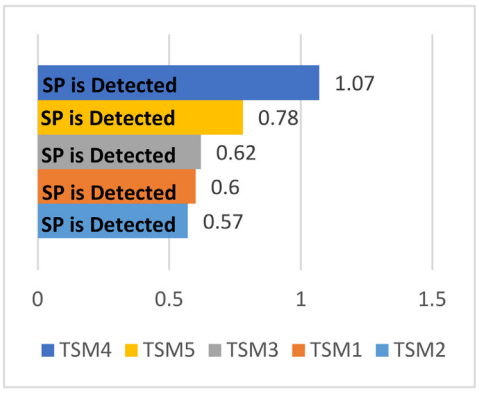

Fig. 20. Different TSMs vs. total time elapsed (s) in the NDMM technique.

problems; however, security patterns are used to solve security-related issues (Open Security Architecture, 2007). They do have some level of similarity based on their documentation and template. Our work is to detect security patterns using matrices as an intermediate language. Our detection approach is different from other design pattern detection techniques. Additionally, in the area of security pattern, no one uses any detection method using matrices. We present some selected research on the detection of design patterns utilizing matrices in their intermediate representation. Some other design pattern detection methods are also available; however, we confine our discussion only to those who use the matrix representation in internal processes, except the work of Bunke and Sohr (2011), which will be discussed in Section 6.2.

6.1. Matrix-based matching detection for design patterns

The detection of design patterns was the early attempt to discover a pattern in the software model or source code by static

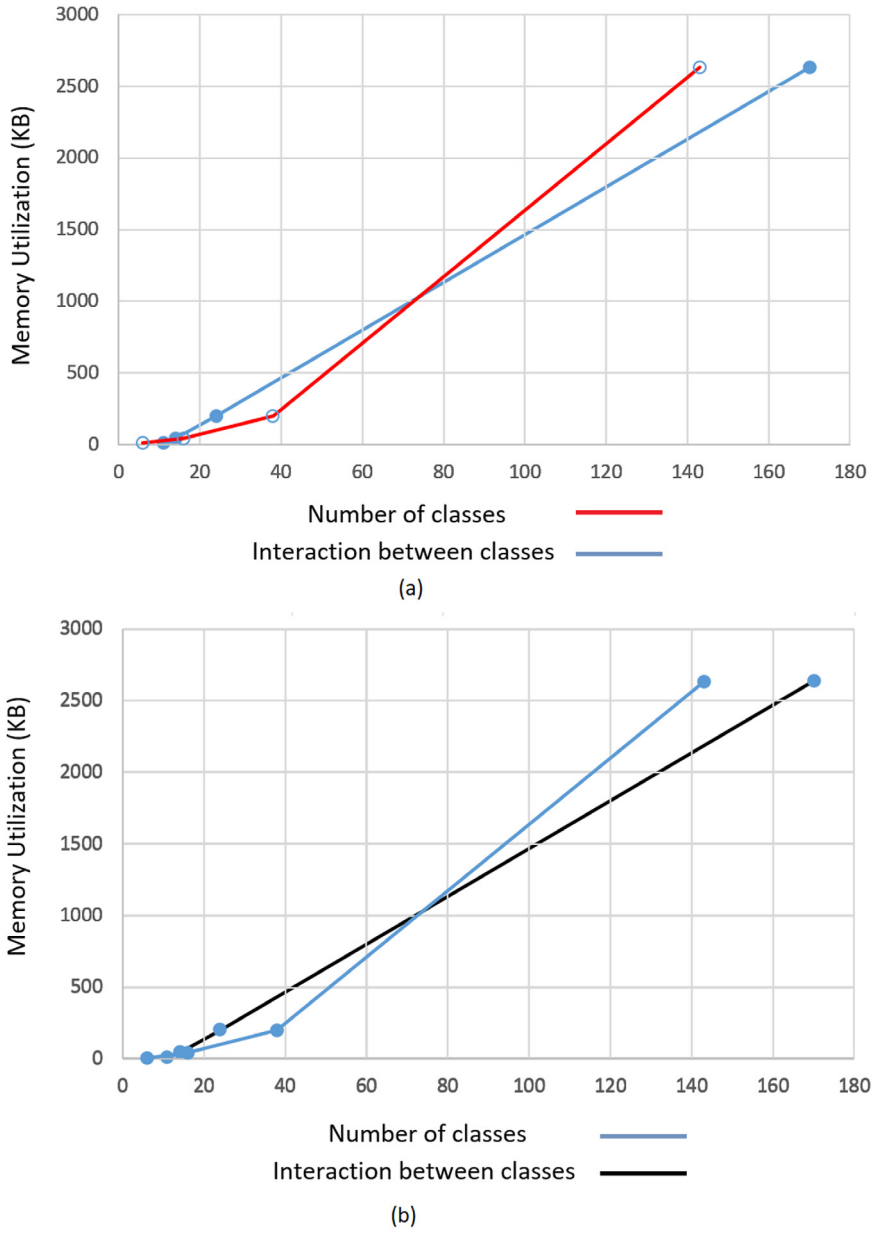

Fig. 21. The number of classes and their interactions vs. memory utilization in (a) the OMM and (b) the NDMM techniques.

analysis or other methods. For example, Tsantalis et al. (2006) propose the detection of design patterns using similarity scoring between graph vertices. They employ the representation of patterns and systems in a matrix form to compute the similarity scoring. The detection method can recognize modified patterns from their standard form. It reduces the size of subject graphs, does not rely on any pattern-specific heuristic, and facilitates extension for original design structures. However, the method only uses structural analysis and does not employ dynamic information for improving further accuracy. The increase in size and the number of classes per package in each subsystem increases the time of convergence of the similarity scoring algorithm. Additionally, the similarity scoring algorithm may take significant memory for a large software system.

Dong et al. (2008b) come up with a solution to calculate the similarity score by cross-correlations for the template matching method to detect design patterns from a software system. They use matrices for the calculation of normalized cross-correlation. They identify exact and variant design patterns from the source code of the software system. The limitation in this approach is that when a new feature is added in a design pattern documentation, the detection method has to be updated accordingly to detect patterns correctly. Compared to Tsantalis et al. (2006), this approach estimates the similarity between the subgraphs of two graphs instead of a pair of vertices.

The use of a matrix to determine the existence of design patterns inside a software system is introduced by Dong and Zhao (2007) and Dong et al. (2007, 2009). They present the detection of design patterns using matrix and weights and develop a toolkit called DP-Miner. In DP-Miner, they apply structural, behavioral, and semantic analysis for detection. The DP-Miner tool builds a matrix of a target system. It encodes the class information into prime numbers or a product of prime numbers, and finally, the design pattern is encoded into the corresponding matrix and weights. The tool uses XMI-based intermediate representations for structural analysis. Behavioral and semantic analyses are implemented to reduce false positives. Further, Dong et al. (2008a) propose the design pattern detection method that incorporates matrix transformations to the cluster. They apply decision tree algorithms to the learning problems consisting of compound records for design pattern detection. The algorithm works by learning the classification rule for composite training examples. It is capable of detecting design pattern instances by learning from training examples developed for a software system.

Gupta et al. (2010a) and Pande et al. (2010a) develop the discovery of patterns by executing a system graph and a design pattern graph (as a template) in a matrix representation for using the normalized cross-correlation approach. The normalized cross-correlation algorithm calculates the degree of similarity of one graph inside the same size or larger graph. This approach has a higher execution time because of computing every relationship matrix distinctively. Also, Gupta et al. (2010b) decompose a graph matching process into phases. The number of phases depends on the number of vertices of the two graphs in matrix depictions. The benefits of this approach are reduced search space and the provision of optimal results. However, it depends on the graphs' size and affects the complexity of the algorithm. Pande et al. (2010c) decompose a target system graph and apply isomorphism using a design pattern graph. They use a set of decomposed components of a target system and a design pattern graph for exploiting isomorphism. Graph decomposition reduces the complexity of a target system graph, and only one permutation matrix of a design pattern is enough for isomorphism. However, in the case of a large number of nodes in a target system, decomposition itself becomes complicated because of a time exhaustion process followed by a matching process in isomorphism.

Pande et al. (2010b) propose a decision tree to develop subgraphs and exhaust graph isomorphism to detect the matched design patterns. The algorithm shows that only a target system of three vertices has six permutation matrices and sixteen vertices in a decision tree graph, whereas, practically, target system vertices are much higher in number. The size of a decision tree graph increases exponentially with vertices. Therefore, the complexity of the detection method rises that causes increased execution time.

Yu et al. (2015) present an approach to the detection of design patterns (use decorator pattern) based on graph isomorphism. They use the graphs of system design and design patterns. They identify candidate classes in the system graph corresponding to pattern classes. They form the subgraphs of candidate classes and apply isomorphism with a pattern graph. The detected isomorphic subgraphs are considered instances of the design patterns. It depends on the filtering process of candidate classes. If the filter could not select the right candidate class, then the isomorphism cannot work to detect instances of the design pattern. Yu et al. (2015) exhibit an approach for the detection of 32 design patterns (such as Adapter, Bridge, Composite, Decorator and Proxy) by subgraph mining and merging using source code. They transform system source code and predefined patterns into graphs where nodes and edges are classes and relationships. They use instances of sub-design patterns to detect inside the system graph. They merge sub-patterns by joint classes and checks the collective matches with selected predefined design patterns. This approach's constraints are the number of detected sub-patterns and their possible number of combinations for matching.

6.2. Detection method for security patterns

Bunke and Sohr (2011) show the detection of the security patterns. They employ a reverse engineering tool suite called Bauhaus (Plödereder et al., 1999) to detect the Single Access Point security pattern in Java-based software systems. They use resource flow graphs provided by the Bauhaus tool as an intermediate representation for software systems. The resource flow graph enables them to utilize the hierarchical reflexion method for analysis. They employ the definition of the Single Access Point pattern to develop a hypothetical architecture as a graph for the static analysis of a target system. They do not discuss false positives in their results because they already map software components utilizing their related names to the Single Access Point component. The detection of security pattern research is not mature. According to a literature survey, only Bunke and Sohr (2011) and Alvi and Zulkernine (2017) attempted to detect security patterns. We propose and implement a security pattern detection (SPD) framework and compare the advantages and disadvantages of matching techniques within the framework.

7. Threats to validity

The following three issues have effects on a security pattern matrix size and value. Hence, the variations of the security pattern matrix are possible and may have an impact on the detection process. Nonetheless, the overall security pattern should fulfill its purpose and will not deviate from the actual security pattern or its variants.

7.1. Variations in the security pattern matrix

We use both UML class and sequence diagrams to understand the relationships between security pattern classes. These diagrams provide general information, and they are not rigid for specific software requirements. They can be modified based

on the software environment, keeping its core idea intact with implementation. For example, for the SAP pattern in Fig. 4(a), the class diagram shows the bidirectional association relationship to all classes. Nevertheless, all relationships are not necessarily bidirectional associations. Furthermore, in Fig. 4(b), the sequence diagram is used to verify or adjust relationships among pattern classes. However, the sequence diagram reveals that the Single Access Point class object has no communication with the object of the Protected System class. In contrast, the relationship between both classes is shown in the class diagram. Therefore, the decision is left to the developer's discretion in using this information and implementing the SAP security pattern in a target system according to software requirements. Nonetheless, the general understanding is evident with the SAP security pattern.

7.2. Security pattern matrix size

The number of classes in a security pattern participating in the detection may vary by considering the functionality of a target system. For example, in Fig. 4(a), the Client class role is not fixed. It may be a human or class or a system that interacts with the software system. Therefore, when we consider Client as class or human, it has a significant impact on the size of a security pattern matrix. The size of the SAP security pattern matrix is 2 x 2 if Client is conceived as an actor (human), and it is a 3 x 3 if it is viewed as a class. Consequently, the extraction of the security pattern matrix depends not only on the class and sequence diagrams but also on the understanding of the target system's functionality.

7.3. Missing class relationship

The SPD tool depends on the quality of the inputs. We prepare a class model of target systems applying reverse engineering on their Java-based source code. The Java-Editor tool is used for reverse engineering, which is the same tool we use to develop the UML class model for the SSS target system. The data extraction process in the SPD tool extracts the class-to-class relationship information from the UML model file (developed using Java-Editor (Röhner, 2018)). Therefore, to ensure the UML class model's quality, we manually analyze the target system source code based on class relationships. Interestingly, class relationship analysis finds a lost connection in the source code of the Simple Android Instant Messaging (SAIM) target system. For example, in the SAIM target system, a relationship does not exist in the UML model between the class com.mekya.IMService (part of the protected system) to the class com.mekya.Login (part of SAP). However, in the source code, we find the association relationship between these classes.

8. Conclusion and future work

Standard security properties, such as confidentiality, integrity, and availability, are required to increase a software system's security. The security patterns address the maintenance of security properties in the software system, provided the patterns are correctly implemented. Our proposed security pattern detection (SPD) framework provides a platform for data extraction, matrix matching, and semantic analysis. We implement two matching techniques, i.e., the ordered matrix matching (OMM) and the non-uniform distributed matrix matching (NDMM). We implement them in the detection framework and compare them to each other.

The OMM technique is based on the detection of an exact submatrix inside a target system matrix. The OMM-based SPD tool shows no false positives. We introduce a semantic analysis to reduce false positives to zero. For semantic analysis, we prepare a multi-level security dictionary (MSPD). The N-gram model of natural language processing (NLP) is used for developing the MSPD, where $N = 2$. The MSPD is developed based on security requirements and security pattern documentations. The OMM technique's overall efficiency is significantly lower concerning computational time for detecting an actual security pattern; however, it may go through all the $n!$ TSMs. The OMM technique provides the location of the security pattern inside the target system.

The NDMM technique works based on class-to-class relationship matching. The uniqueness of the NDMM technique is detecting a security pattern from any one of the $n!$ TSMs, where n is the number of target system classes (provided that a security pattern is available inside the target system). Also, most of the time, the NDMM technique uses slightly less memory and computational time than the OMM technique for the detection process of the same target systems. In the case of false-positive results, we employ the semantic analysis technique to reduce false positives. The NDMM technique detects the location of the security pattern in the source code of the target system. We evaluate both matching techniques concerning complexity, execution time, accuracy, and the security pattern's location. We find that the NDMM technique is intrinsically better than the OMM technique.

In the future, a diagonally distributed matrix matching technique may be considered to develop a powerful and fast technique for security pattern detection. Further, the multi-level security pattern dictionary can be extended for other computer languages developed based on non-English alphabets or have other rules for naming convention. Besides, we can fine-tune the detection process for security patterns by considering all kinds of class to class relationships such as inheritance, aggregation, and composition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

Alexander, C., Ishikawa, S., Silverstein, M., 1977. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, New York.

Alvi, A.K., Zulkernine, M., 2011. A natural classification scheme for software security patterns. In: Proceedings of the 9th International IEEE Conference on Dependable, Autonomic and Secure Computing (DASC), Sydney, Australia, pp. 113–120.

Alvi, A.K., Zulkernine, M., 2012. A comparative study of software security pattern classifications. In: Proceedings of 7th International IEEE Conference on Availability, Reliability, and Security (AReS), Prague, Czech Republic, pp. 582-589.

Alvi, A.K., Zulkernine, M., 2017. Security pattern detection using ordered matrix matching technique. In: Proceedings of 3rd International IEEE Conference on Software Security and Assurance (ICSSA); July 11, Altoona, Pennsylvania, USA.

Binkley, D., Davis, M., Lawrie, D., Morrell, C., 2009. To camel case or under-score. In: Proceedings of IEEE 17th International Conference on Program Comprehension (ICPC), Vancouver, BC, Canada, pp. 158-167.

Bjork, R.C., 2019. ATM simulation. In: Professor of Computer Science. Gordon College, Wenham, MA, U.S.A., p. 01984, http://www.math-cs.gordon.edu/courses/cs211/ATMExample/index.html. Developed in 1996. (accessed 21 January 2019).

Bunke, M., Sohr, K., 2011. An architecture-centric approach to detecting security patterns in software. In: Ulfar, E., Roel, W., Nicola, Z. (Eds.), Proceedings of 3rd International Conference on Engineering Secure Software and Systems (ESSoS). Springer-Verlag, Berlin, Heidelberg, pp. 156–166.

Campwood Software, 2018. Sourcemonitor 3.5.6. http://www.campwoodsw.com/sourcemonitor.html. (accessed 30 2018).

Dong, J., Lad, D.S., Zhao, Y., 2007. DP-miner: Design pattern discovery using matrix. In: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS). IEEE Computer Society, Washington, DC, USA, pp. 371–380.

Dong, J., Sun, Y., Zhao, Y., 2008a. Compound record clustering algorithm for design pattern detection by decision tree learning. In: Proceedings of IEEE International Conference on Information Reuse and Integration (IRI), Las Vegas, NV, USA, pp. 226-231.

Dong, J., Sun, Y., Zhao, Y., 2008b. Design pattern detection by template matching. In: Proceedings of the ACM Symposium on Applied Computing. ACM, New York, NY, USA.

Dong, J., Zhao, Y., 2007. Experiments on design pattern discovery. In: Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering. IEEE Computer Society, Washington, DC, USA, p. 12.

Dong, J., Zhao, Y., Sun, Y., 2009. A matrix-based approach to recovering design patterns. IEEE Trans. Syst. Man Cybern. 39 (6), 1271–1282. http://dx.doi.org/10.1109/TSMCA.2009.2028012.

Eden, A., Gasparis, E., Nicholson, J., et al., 2018. Round-trip engineering with the two-tier programming toolkit. Softw. Qual. J. 26 (2), 249–271. http://dx.doi.org/10.1007/s11219-017-9363-9.

Fernandez, E.B., 2013. Security Patterns in Practice: Designing Secure Architectures using Software Patterns. John Wiley & Sons, New York.

Fernandez, E.B., Washizaki, H., Yoshioka, N., Kubo, A., Fukazawa, Y., 2008. Classifying security patterns. In: Zhang, Y., Yu, G., Bertino, E., Xu, G. (Eds.), Proceedings of 10th Asia-Pacific Web Conference (APWeb), Progress in WWW Research and Development. In: Lecture Notes in Computer Science, vol. 4976, Springer, Shenyang, China, Berlin, Heidelberg, pp. 342–347. http://dx.doi.org/10.1007/978-3-540-78849-2_35.

Fig, M., 2016. File exchange: findsubmat, MATLAB central. https://www.mathworks.com/matlabcentral/fileexchange/23998-findsubmat. (accessed 27 July 2016).

Gamma, E., Helm, R., Johnson, R., Vlissides, J.M., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Upper Saddle River, Addison-Wesley Professional, New Jersey.

Gorton, I., 2011. Software quality attributes. In: Gorton, Ian (Ed.), Essential Software Architecture. Springer, Berlin, Heidelberg, pp. 23–38. http://dx.doi.org/10.1007/978-3-642-19176-3.

Gupta, M., Pande, A., Rao, R.S., Tripathi, A.K., 2010. Design pattern detection by normalized cross-correlation, In: Proceedings of International Conference on Methods and Models in Computer Science (ICM2CS), New Delhi, India, pp .81-84 http://dx.doi.org/10.1109/ICM2CS.2010.5706723, (accessed 05 May 2017).

Gupta, M., Rao, R.S., Tripathi, A.K., 2010. Design pattern detection using inexact graph matching. In: Proceedings of International Conference on Communication and Computational Intelligence, Erode, India, pp. 211-217 https://ieeexplore.ieee.org/document/5738733 (accessed 03 April 2017).

Hafiz, M., Adamczyk, P., Johnson, R., 2011. Growing a pattern language (for security). In: Proceedings of 18th Conference on Pattern Languages of Programming (PLOP), Portland, Oregon, USA, pp. 139-158, http://dx.doi.org/10.1145/2384592.2384607.

Jones, D.M., 2019. The new C Standard: An economic and cultural commentary. http://www.coding-guidelines.com/cbook/sent792.pdf. (accessed 22 2019).

Kienzle, D.M., Elder, M.C., 2019. Final technical report: Security patterns for web application development. In: Defense Advanced Research Projects Agency, DARPA Contract. F30602-01-C-0164. http://www.scrypt.net/~celer/securitypatterns/finalreport.pdf. Published 30 2002. (accessed 13 February 2019.

Kienzle, D.M., Elder, M.C., Tyree, D., Edwards-Hewitt, J., 2019. Security patterns repository. version 1.0. http://www.scrypt.net/~celer/securitypatterns/repository.pdf. 2002. (accessed 25 April 2019.

Konagaya, M., Otachi, Y., Uehara, R., 2014. Polynomial-time algorithms for subgraph isomorphism in small graph classes of perfect graphs. In: Gopal, T.V., Agrawal, M., Li, A., Cooper, S.B. (Eds.), Proceedings of International Conference on Theory and Applications of Models of Computation (TAMC2014), Vol. 8402. In: Lecture Notes in Computer Science, (1), Springer, Chennai, India, Cham, Switzerland, pp. 216–228. http://dx.doi.org/10.1007/978-3-319-06089-7_15.

Lee, J., Han, W.S., Kasperovics, R., Lee, J.-H., 2012. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In: Proceedings of the VLDB Endowment, Vol. 6. (2), pp. 133–144. http://dx.doi.org/10.14778/2535568.2448946.

Mermerkaya, A.O., 2013. Simple android instant messaging application. https://code.google.com/archive/p/simple-android-instant-messaging-application/downloads. Updated Jul 14, 2013. (accessed 29 2018).

Meszaros, G., Doble, J., 1997. A pattern language for pattern writing. In: Martin, R.C., Riehle, D., Buschmann, F. (Eds.), Pattern Languages of Program Design, Vol. 3. Addison-Wesley Longman Publishing Co., Boston, Massachusetts, pp. 529–574.

Open Security Architecture, 2007. IT Security Patterns. https://www.opensecurityarchitecture.org/cms/definitions/security_patterns (accessed February 2020).

Pande, A., Gupta, M., Tripathi, A.K., 2010a. Design pattern mining for GIS application using graph matching techniques. In: Proceedings of 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT), Chengdu, India, pp. 477-482, https://ieeexplore.ieee.org/document/5564518. Accessed 2018.

Pande, A., Gupta, M., Tripathi, A.K., 2010b. A decision tree approach for design patterns detection by subgraph isomorphism. In: Das, V.V., Vijaykumar, R. (Eds.), Proceedings of International Conference Information and Communication Technologies (ICT). In: Information and Communication Technologies. Communications in Computer and Information Science, vol. 101, Springer, Kochi, Kerala, India, Berlin, Heidelberg, http://dx.doi.org/10.1007/978-3-642-15766-0_95.

Pande, A., Gupta, M., Tripathi, A.K., 2010c. A new approach for detecting design patterns by graph decomposition and graph isomorphism. In: Ranka, S., et al. (Eds.), Proceedings of 3rd International Conference on Contemporary Computing (IC3). In: Contemporary Computing. Communications in Computer and Information Science, vol. 95, Springer, Noida, India, Berlin, Heidelberg, http://dx.doi.org/10.1007/978-3-642-14825-5_10.

Plödereder, E., Koschke, R., Wittiger, M., et al., 1999. Project Bauhaus, in Collaboration with the University of Stuttgart. University of Bremen, and Axivion GmbH, Stuttgart, Germany, https://www.iste.unistuttgart.de/ps/research/projects/index.html #id–3333f339-0. Access on 13 2019.

Röhner, G., 2018. Java-editor, version 15.26, Germany. http://javaeditor.org/doku.php. (accessed 7 August 2018).

Rossum, G., Warsaw, B., Coghlan, N., 2013. PEP 8 – style guide for python code, published. https://www.python.org/dev/peps/pep-0008/. (accessed 27 August 2017).

Sandler, D.R., Derr, K., Wallach, D.S., 2008. VoteBox: A tamper-evident, verifiable electronic voting system. In: Proceedings of the Seventeenth USENIX Security Symposium. http://votebox.cs.rice.edu. (accessed 2 July 2018).

Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P., 2006. Security Patterns: Integrating Security and Systems Engineering. John Wiley & Sons Ltd., Chichester, West Sussex, England.

Slavin, R., Niu, J., 2018. Security Patterns Repository. Department of Computer Science. The University of Texas at San Antonio, San Antonio, Texas, USA, http://sefm.cs.utsa.edu/repository/ (accessed 25 2018).

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S., 2006. Design pattern detection using similarity scoring. IEEE Trans. Softw. Eng. 32 (11), 896–909.

Yskout, K., Heyman, T., Scandariato, R., Joosen, W., 2006. A System of Security Patterns. Technical Report# CW469, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, https://www.researchgate.net/publication/242679421_A_system_of_security_patterns. (accessed 21 March 2019).

Yu, D., Zhang, Y., Chen, Z., 2015. A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. J. Syst. Softw. 103 (C), 1–16. http://dx.doi.org/10.1016/j.jss.2015.01.019.

Aleem Khalid Alvi is a doctoral candidate and research assistant in the School of Computing of Queen's University, Canada. He is a member of the Queen's Reliable Software Technology (QRST) research group. His research interests include software security, cloud security, network security, cryptography and steganography. Mr. Alvi received a Bachelor of Science (Math) from the University of Karachi, Bachelor of Engineering (Elect.) and Master of Science (Elect.) from NED University of Engineering & Technology, Karachi, Pakistan. His research contribution is visible at https://research.cs.queensu.ca/home/aleem/.

Dr. Mohammad Zulkernine is a Professor and Canada Research Chair in the School of Computing of Queen's University, Canada, where he leads the Queen's Reliable Software Technology (QRST) research group. Dr. Zulkernine received his Ph.D. from University Waterloo, Canada. He joined Queen's in 2003 and spent his sabbatical as a visiting professor at the University of Trento, Italy and as a researcher at Irdeto Canada. Dr. Zulkernine is a senior member of the IEEE and the ACM, and a licensed professional engineer in the province of Ontario, Canada. His current research focuses on building reliable and secure software systems, and he has extensive publications in this area. He has led major research projects supported by a number of provincial and federal agencies and industry partners. He has been at leadership positions such as a general chair, organizing chair and program chair of many major research conferences and workshops. More information about Dr. Zulkernine is available at http://research.cs.queensu.ca/home/mzulker/.