Hindawi
Wireless Communications and Mobile Computing
Volume 2022, Article ID 1657627, 11 pages
https://doi.org/10.1155/2022/1657627

Research Article

An Approach for Verification of Secure Access Control Using Security Pattern

Charu Gupta, Rakesh Kumar Singh, and Amar Kumar Mohapatra

Department of Information Technology, Indira Gandhi Delhi Technical University for Women, Delhi 110006, India

Correspondence should be addressed to Charu Gupta; charugupta@igdtuw.ac.in

Received 15 June 2022; Accepted 12 August 2022; Published 6 September 2022

Academic Editor: Kapil Sharma

According to OWASP-2021, more than 3,00,000 web applications have been detected for unauthenticated and unauthorised access leading to a breach of security trust. Security patterns are commonly used in web applications to address the problem of broken access. Web developers are not experts in implementing security patterns. Therefore, it is necessary to verify that the security pattern has been applied, specifying the original intent of the security pattern. In this paper, an approach has been proposed that analyses the behavioural aspect of security patterns to verify that it meets the security requirement of the web application. The proposed approach extracts the class diagram's structural properties, relations, associations, and security-related constraints and verifies it using the first-order predicate logic. Experiments have been conducted using class diagrams of security patterns to detect instances of broken access control early in the design phase. The proposed approach will help minimise the risk of unauthenticated and unauthorised access to a web application.

1. Introduction

According to OWASP-2021, security vulnerabilities causing broken access control have moved to the first position from fifth in the last three years [1]. 94.5% of web applications have been detected with security weaknesses causing unauthorised disclosure, distortion, disruption, or data destruction by allowing users to perform actions, not within their respective limits [1]. Such weaknesses of broken authentication and unauthorised access have been reported in more than 318,000 web applications [1]. The percentage of vulnerabilities due to broken access control is increasing [2], as shown in Figure 1. These vulnerabilities exist in web applications due to inadequate design, hardcoding of access control and rights, and overlooking security best practices during the software development life cycle [3]. Moreover, frequent changes in software applications to patch security vulnerabilities bring new challenges for testing and removing the bugs [4–6].

Security patterns are commonly used in web applications and framework to address the problem of broken access [7]. In literature, the design and description of security patterns are commonly characterised by classes consisting of an interface class, abstract class, method, data members, and their respective implementations in concrete classes. The UML description, code examples, and design recovery regarding security patterns are not yet as mature as their counterparts in the software design patterns [8]. The organisation, structure, relationships, and dependencies of member data, methods, and classes in a security pattern represent a recurring design concept that can be utilised to verify and validate before implementation [9]. A sample code may not accompany the security pattern description for implementation. Therefore, during the software design phase, the class diagram of the security patterns requires a thorough examination to vouch for any unintentional security breach or hidden vulnerability. This vulnerability may be caused at run time due to the establishment of undesired connections among various objects of a class diagram. An attacker may exploit such a connection and access confidential data with time. The security patterns, thus applied, need to be verified for their consistency and usability. The class diagram in the design document provides a structural and behaviour view of member elements and functions. If not

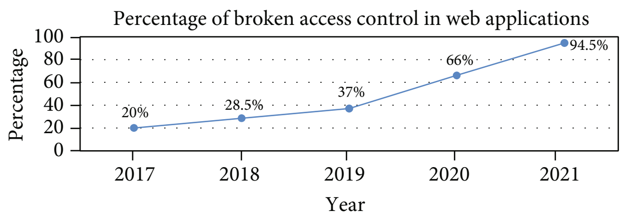


Figure 1: Increasing trend of broken access control detected in web applications in the last five years [1].

applied appropriately, the properties of relationships and accessibility lead to unauthorised access to resources by the users. This leads to a hidden vulnerability that goes unnoticed until the attacker exploits the vulnerability. For example, consider an operation update op1 on a data element $d1$ is meant only for a role R_x.

In contrast, other operation read op2 on the same data element $d1$ is meant for various R_x, R_y, and R_z roles. The secure visitor pattern is applied to ensure role-based data access. However, suppose the accessibility (public, private, protected, and friend) of data elements in abstract and concrete classes is not defined appropriately. In that case, it will allow restricted roles to access the data elements and operations for which the role is not authorised.

Moreover, certain security constraints are not shown in the class diagram but are written in textual statements in security patterns. The textual statements that could not be depicted in the class diagram also go unnoticed during the implementation of the security pattern. These security issues leave vulnerability that goes unnoticed even during rigorous security testing. The attackers take advantage of these hidden security vulnerabilities and take over resource control.

This paper proposes an approach to verify the relationships and accessibility among various objects created during the execution of a security pattern. The proposed approach extracts the microarchitecture from the class diagram of the security pattern applied. The security constraints specified in the security pattern description are written in first-order predicate logic. The proposed approach has been applied to secure visitor, secure strategy factory, and authenticator patterns to check the consistency of relationships among various concrete elements of the abstract security pattern. The extracted microarchitecture and security constraints are analysed by generating their instances using Alloy. The experiments show that connections indicating unauthorised access and broken access control are detected in one or more instances. The detected instance is rectified using the proposed approach, and a metamodel is generated for the security pattern. The approach will facilitate the appropriate implementation of security-dependent logic in a web application and helps in the identification of hidden vulnerabilities at an early stage.

The rest of the paper is organised into four sections describing Related Work, Proposed Methodology, Experimental Results, Analysis, and Discussion, and Conclusion and Future Work.

2. Related Work

In this section, various approaches for verifying and validating security patterns have been discussed. Dong et al. [10] represented the composition of security patterns using Calculus of Communicating System- (CCS-) based model checker of sequence diagrams. The approach proposed by Dong et al. [10] verified the states and its transitions in the sequence diagram of a security pattern. Mourad et al. [11] proposed an aspect-oriented two-phased approach to verify integrated security patterns in an application. The approach defines security objects, methods, and events and manually verifies them with security requirements. Pedroza et al. [12] proposed a verification approach using block and state machine diagrams in the SysML environment for the safety and security of critical real-time embedded systems.

Heyman et al. [13] modelled and verified compositional and trust properties of a software security architecture using security patterns based on abstract and concrete levels. Castellanos et al. [14] presented a verification method of security and dependability patterns in preconditions, postconditions, and model transformations. Devyanin et al. [15] performed a data flow analysis of the operating system security model using role-based access control to prove its conformance and consistency to ensure integrity and confidentiality. Vaca and Gasca [16] defined security patterns using Feature-Oriented Domain Analysis (FODA) and represented mandatory forces of security patterns in the Backus Naur Form. Hamid et al. [17] proposed a semiformal approach for representing security patterns at domain-independent, domain-specific, and pattern-specific metamodel. Alzahrani [18] demonstrated the use of codecharts for formal specification and verification of security patterns in terms of generalisation and abstraction. Dwivedi and Rath [19, 20] presented verification of five security patterns of web applications and one security pattern of Service-Oriented Architecture (SOA) using Alloy [21]. The security patterns verified by Dwivedi and Rath [19, 20] are secure proxy, single-sign-on, check point, authenticator, and access policy.

He and Fu [22] modelled and analysed six security patterns, namely, account lockout, authenticated session, client data storage, encrypted storage, password authentication, and password propagation, to check their completeness, consistency, and ambiguity in their textual descriptions. He and Fu [22] used high-level petri nets to ensure the correct implementation of these six security patterns. Near and Jackson [23] showed that previously unknown security bugs could be easily identified using their proposed formal approach SPACE (Security Pattern Checker), which finds implementation bugs in access control security patterns. In an approach to improve security pattern definition, Beherens [24] provided abstractions and their implementations using formalised notation. The constraints were represented as a finite state machine recognised by regular language for analysis and verification.

Berghe et al. [25] focused on defining security patterns using a modelling language and proposed four data-specific building blocks, namely, data types, data flows, data creation, and data storage, to support security patterns. Security analysis of the web in terms of cache usage, temporal logic, and state transitions was presented by Shimamoto et al. [26]. Shimamoto et al. [26] also verified Web Deception attack, CSRF attack, and exact origin and cross-origin Browser Cache Poisoning (BCP) attack using Alloy and temporal logic syntax. Obeid and Dhaussy [27] presented a message, resource, and access-based approach for formalisation, verification, and composition of security patterns with increased complexity measures.

Gadouche et al. [28] used Event-B correct-by-construction methodology to specify declarative and behavioural aspects of Role-Based Access Control. Gabillion et al. [29] designed a model for representing dynamic and contextual authorisation rules using first-order predicate logic for security administration and policy in the Internet of Things. Gupta et al. [30] proposed a formal approach to represent security constraints of a security pattern using first-order predicate logic. The formal specification facilitated early detection of hidden security vulnerabilities in a software or a web application.

The approaches available in the literature for specification and verification of security patterns are based on transitions and a set of actions and cover few security patterns. The existing approaches have considered transactions, temporal logic, request, response, and resource messages to verify the security of an application. As the application grows in size bringing variations in code, the greater efforts are required to verify security properties and detect vulnerabilities [31]. However, the existing approaches grow exponentially as the number of transitions and states becomes larger in terms of complexity and execution time. The relationships and security-related constraints among various objects created in the execution of security patterns have not been analysed in existing approaches. The existing approaches have not verified the behavioural aspect of security patterns among various instances of concrete objects.

3. Proposed Approach

This paper proposes an approach to verify secure access control and detect unauthenticated and unauthorised access arising from the inadequate implementation of security patterns. The class diagram of the security pattern available in literature has been used to extract its microarchitecture in the proposed approach. The microarchitecture of the class diagram contains structural properties such as interface class, abstract class, concrete class, methods, fields, and their respective relations and accessibility. An interface class is identified along with its member functions, data, and accessibility specifiers. The approach then identifies the abstract class and its concrete implementations. The private and protected data fields and member functions are identified for every concrete class. The private and protected member elements are the restricted elements to be verified for their non-accessibility from other elements. Subsequently, other member elements with public and friend accessibility are also identified and checked for the parameters passed and consistency. The interface, abstract, and concrete classes are analysed for the relationships of inheritance, association, composition, aggregation, and creation of objects.

Further, security constraints are written in first-order predicate logic and modelled with the microarchitecture. For example, suppose in a security pattern, it is defined that a concrete class should not have a public method. In that case, it is written as $\nexists \text{ispublic}(\text{operations}(\text{ConcreteClass}))$, in which $\nexists$ is a negative existential quantifier of predicate logic. Similarly, a class may have only one member function and the predicate logic for same is $\text{Operations}(\text{ConcreteClass}X) = \{\text{Op}X()\}$. The identified microarchitecture is then analysed and executed using Alloy Specification Language. The security pattern is then executed by creating multiple instances and objects. Each execution is verified for the existence of any counterexample instance in its respective instance. If a counterexample is detected, the constraints and structure are rectified suitably to present an accurate design of security pattern. All nodes in an instance are checked for reachability [32, 33] to verify the complete and unambiguous implementation of the security pattern. The proposed approach for verifying applied security patterns and identifying any hidden vulnerabilities is shown in Algorithm 1.

4. Experiments, Results, and Analysis

The proposed approach has been applied to three security patterns: authenticator, secure visitor, and secure strategy. The microarchitecture has been extracted and detailed in Section 4.1 for each applied security pattern in the first step. The subsequent modelling, execution of multiple instances, results, and analysis for each security pattern have been discussed.

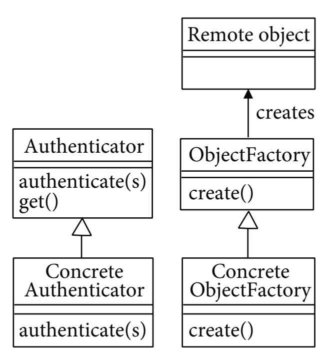
4.1. Extracting Microarchitecture of Security Patterns

4.1.1. Authenticator Pattern. The class diagram of the authenticator pattern [34] is shown in Figure 2(a). The microarchitecture from the class diagram of the authenticator pattern is extracted by identifying its classes, member functions, security constraints, and relations and is shown in Figures 2(b) and 2(c).

4.1.2. Secure Visitor Pattern. The secure visitor pattern class diagram [35] is shown in Figure 3(a). The secure visitor pattern separates conditional security logic from the business logic in hierarchical data nodes in a web application. It enables data nodes to authenticate the visitors requesting their access for certain operations. The usage of the secure visitor pattern helps prevent unauthorised access to data by implementing security logic in a separate code segment or class. The secure visitor pattern provides a solution to prevent such attacks by making data nodes lock themselves from being read by a visitor unless the visitor supplies the proper credentials to unlock the data node. The secure visitor pattern consists of an interface, abstract, and concrete classes representing secure visitors, unlocked and locked data nodes, various member data and functions, and their respective access specifiers. The abstract

```
Input    class diagram of security pattern
Process
Step 1    extraction of microarchitecture
   For each security pattern
   Identify classes, member functions, and member data from the class diagram.
      For each class identified, label it as an interface, abstract class, or concrete class.
         For each function identified, label it as an abstract function or a concrete function.
            For each member, data and function
               Identify the accessibility
            End for
         End for
      End for
   Look into the textual description of security patterns to identify security constraints.
      For each security constraint
         Write first-order predicate logic.
      End for
   End for
Step 2    model the security pattern in Alloy using the mapping of extracted microarchitecture, assertion, and facts
Step 3    run and execute the assertion for multiple instances
Step 4    check for counterexample
   If counterexample found
      Rectify the relation, association, and function calls
      Go to step 1
   Else
      Terminate with metamodel
Output    metamodel for security pattern for code implementation
```

Algorithm 1: Algorithm for analysing and identifying vulnerabilities in the implementation of security patterns at the design phase.



(A) Class diagram of authenticator pattern [34]

Authenticator
Authenticator is an interface.

Abstract classes: ObjectFactory, Remote, Authenticator
Abstract operations: get, auth, create

Concrete classes: ObjectA, ObjectB, ConcreteAuthenticatorA, ConcreteAuthenticatorB
Operations: getA, authA, createA, getB, authB, createB
Where A,B are the methods restricted to the different authentication methods

(B) Microarchitecture of authenticator pattern

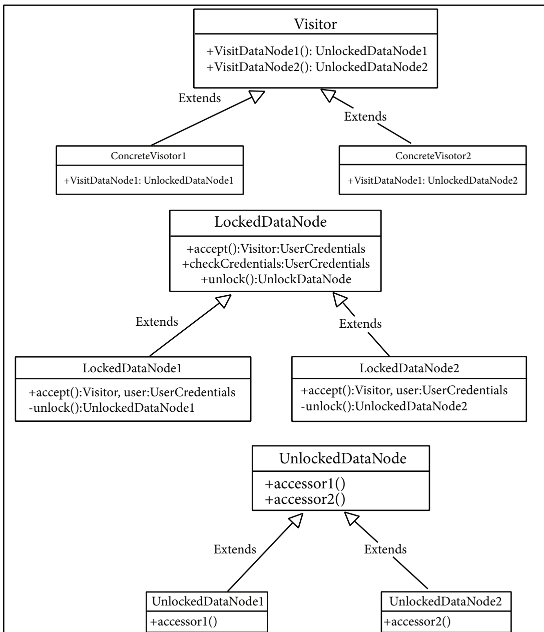
Security constraints in first order predicate logic

all x: Authenticator | all y: ObjectFactory |
one $x \geq y$ && one $y \geq x$ implies one $y \geq x$ ||
create [x, y] implies one $x \geq y$

C) Security constraints of authenticator pattern

Figure 2: Extracting microarchitecture of authenticator pattern.

function 'accept()' is implemented in concrete classes of locked data node type(s) that, in turn, unlock the respective data node after checking a user's credentials. The microarchitecture of the secure visitor pattern extracted is shown in Figure 3(b). Security constraints such as for every locked data node, there should be an unlocked data node, restriction of access to parent and child data nodes, and restrictions on member functions of abstract and concrete classes are represented in predicate logic and shown in Figure 3(c). These security constraints are crucial for providing secure access and preventing broken access in web applications.



(A) Class diagram of secure visitor pattern [34]

SecureVisitor
Secure visitor is an interface

Abstract classes: UnlockedObject, UnlockedDataNode, LockedDataNode, UserAndRolesCredentials
Operations: accept, unlock, checkCredentials, Visit

Concrete classes: VisitorX, UnLockedDataNodeX, LockedDataNodeX
Where X ranges from 1 to *n*.

a) UnlockedObject → UnlockedDataNode ∈ associations
b) relations(UnlockedObject, UnlockedDataNode, associations)
c) accept∈ operations(LockedDataNode)
d) unlock∈ operations(LockedDataNode)
e) accept∉ operations(UnlockedDataNode)
f) CheckCredentials ∈ operations (LockedDataNode)
g) VisitX⊆ operations(SecureVisitor)
h) isProtected(checkCredentials)
i) isPublic(accept)
j) isPrivate(unlock)

(B) Microarchitecture of secure visitor pattern

Security constraints in first order predicate logic

a) Association is from UnlockedObject to UnlockedDataNode
   a. type (UnlockedObject) = Object ∧
   b. type (UnlockedDataNode) = UnlockedDataNode ∧
   c. multiplicity (UnlockedDataNode) = “*”
b) Client depends only on SecureVisitor and UnlockedObject
   a. Such that:
      i. access (SecureVistor, UnlockedObject),
      ii. {UnlockedDataNode} ∪ subclasses (UnlockedDataNode) ∪ subclasses (SecureVisitor)
c) Accessor is the only operation in UnlockedDataNode
      i. Operations (UnlockedDataNode) = {accessor}
d) For every subclass of UnlockedDataNode, there exists a subclass belonging to LockedDataNode and an operation VisitX belonging to the subclass of SecureVistitor. The function visit () takes the parameter UnlockedDataNode of typeX as its subclass such that each concrete node in the data hierarchy has both a locked and unlocked version. Only a locked node will be able to create an unlocked node.

$$\forall x \in \text{DataHierarchy}\ \exists!\, x\,(l, u) \text{ such that}$$
$$\{\exists\, u \in \text{subclasses (UnlockedDataNode)} \wedge \exists\, l \in \text{subclasses (LockedDataNode)} \wedge \text{create}\,(l, u)\}$$

e) No UnlockedDataNode should be able to access its parent or child data node.

$$\forall x \in \text{subclasses (UnlockedDataNode)}\ \sim \text{access}\,(x, x.child) \wedge \sim\text{access}\,(x, x.parent)$$

f) LockedDataNodes has no public operations

$$\nexists\ \text{ispublic (operations (LockedDataNode))}$$

g) To access UnLockedDataNode, visitor should call LockedDataNode, which in return create UnLockedDataNode after checking user credentials.

$$\forall x \in \text{DataHierarchy}$$
$$\exists accept \in \text{operations (LockedDataNodeX)} \wedge$$
$$\exists checkCredentials \in \text{operations (LockedDataNodeX)}$$

such that

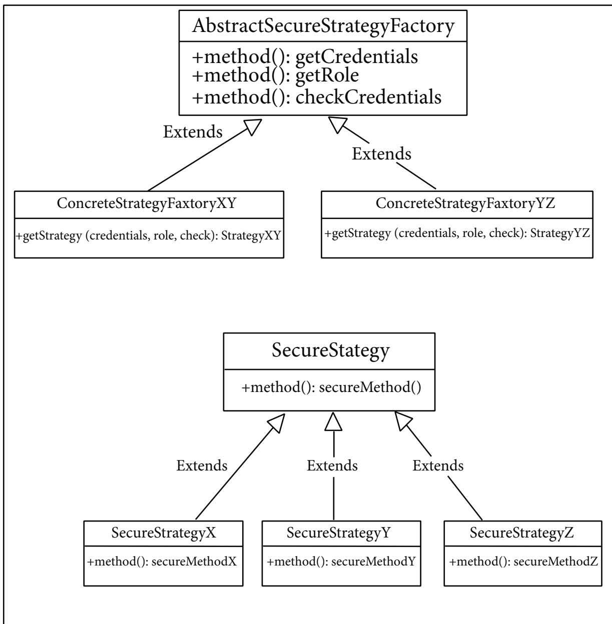
if return(checkCredentials)=true

create (l, u)

C) Security constraints of Secure Visitor Pattern in first order predicate logic

Figure 3: Extracting microarchitecture of secure visitor pattern.

4.1.3. Secure Strategy Factory Pattern. The class diagram of the secure strategy factory pattern [35] is shown in Figure 4(a). The secure strategy factory pattern separates the security-dependent logic associated with each role from the basic functionality of object creation and selection. Secure strategy factory implements the creation and selection of an object for executing an operation depending on a set of security credentials. The given security credentials are used to select and return the role-specific object. The different secure functions are implemented in various concrete



SecureStrategy
SecureStrategy is an interface.

Classes: SecureStrategy, UserAndRolesCredentials, RoleAccess
Operations: getCredentials, getRole, checkCredentials, secureMethod

Classes: secureStrategyX, secureStrategyY, secureStrategyZ
Operations: secureMethodX, secureMethodY, secureMethodZ
Where X,Y,Z are the methods restricted to the roles X,Y,Z

a) secureMethod → secureTrust ∈ associations
b) relations (secureMethod, secureTrust, associations)
c) fetchLevelAccess∈ LevelAccess
d) getCredentials ∈ operations (fetchRoleAccess)
e) checkCredentials ∈ operations(fetchRoleAccess)
f) isProtected (checkCredentials)
g) isPublic (getCredentials)
h) isPrivate (secureMethod)

(A) Class diagram of secure strategy factory pattern [34]

(B) Microarchitecture of secure strategy factory pattern

Security constraints in first-order predicate logic
For every trust level x of LevelAccess, there exists a concrete subclass SecureStrategy that belongs to the role with the method definition of secureMethod.

$$\forall x \in \text{LevelAccess}\ \exists!\, x\,(f, m) \text{ such that } \exists f \in \text{subclasses (SecureStrategy)} \wedge \exists m \in \text{subclasses (SecureStrategy)} \wedge \text{define}\,(f, m, x)$$

C) Security constraints of secure strategy factory pattern in first-order predicate logic

Figure 4: Extracting microarchitecture of secure strategy factory pattern.

implementations of abstract secure strategy factory for each set of roles defined by the security requirements. For example, in a web application, there are three roles having complete, little, or no trust, and then, three concrete implementations of secure strategy will be created, one for each level of trust. A concrete implementation of secure strategy factory will only contain functionality restricted to a secure role or trust level. The microarchitecture of the secure strategy factory pattern is extracted using the proposed approach as in Figure 4(b), and the security constraints written in first-order predicate logic are shown in Figure 4(c).

4.2. Detection of Broken Access Control. In this section, the class diagram of the security pattern is executed in the Alloy tool to detect any violation of security requirements. Each pattern has been executed for multiple instances to create objects, and each instance is analysed for the existence of any counterexample. On detecting a counterexample, the nodes in the graph are analysed for the inappropriate link. The security pattern is then rectified by correcting the definition of concrete classes and methods and implementing the security predicate to verify the security requirement.

4.2.1. Authenticator Pattern. The class diagram of authenticator pattern is executed to detect any hidden vulnerabilities. In Figure 5, while executing two or more concrete authenticators, it is found that concrete authenticator0 is creating an object for the concrete authenticator1. It is detected that the concrete object of concrete authenticator1 is inheriting the abstract method instead of overriding it. It is also detected that an object created by one authenticator can create an object of another authenticator by an inherited method of abstract authenticator. The security error is rectified by ensuring that $\text{pred creates}\,[x : \text{one ConcreteAuthenticator}, y : \text{one Object}A]\{\}$ and overriding the method create() in each concrete authenticator.

4.2.2. Secure Visitor Pattern. The secure visitor pattern class diagram is executed, and hidden vulnerabilities are detected using the proposed approach. The predicate logic for the security constraint is that only a locked data node can create its unlocked data node. However, on the execution of the predicate visit for two or more instances in secure visitor, an instance is found in Figure 6 that shows unlocked data node2 could unlock data node1. The instance creates unauthorised access to data node2 via data node1, against the security requirements. On analysing the microarchitecture, it is detected that the function visit() has been implemented inappropriately.

$$\text{pred visit}\,[v : \text{SecureVisitor}, l : \text{LockedDataNode}]\,\{\,\}. \quad (1)$$

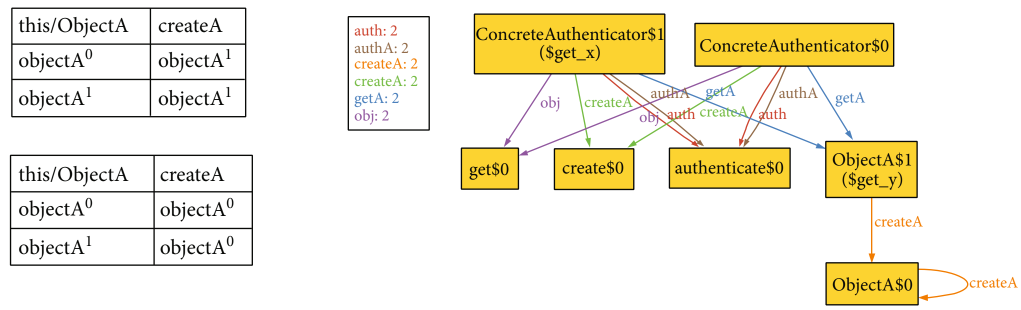


Figure 5: An instance generated for the authenticator pattern showing the creation of multiple objects and broken authentication.

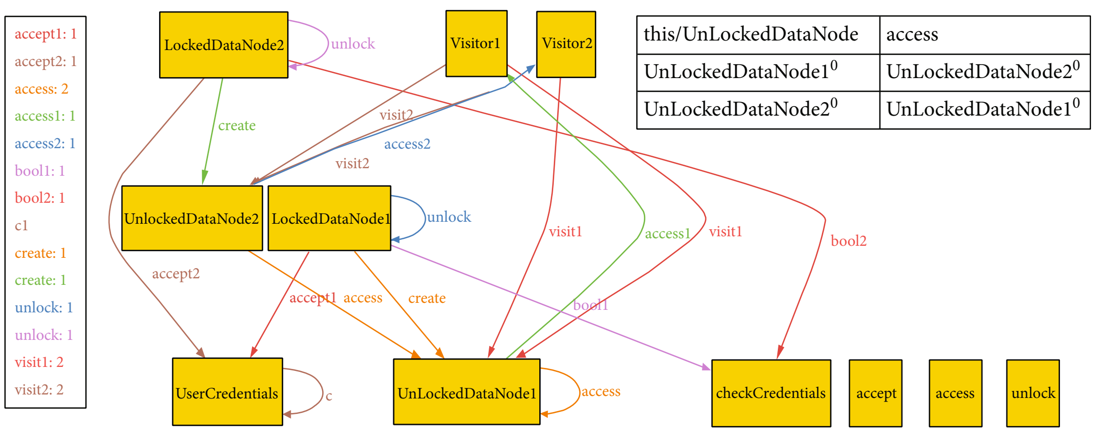


Figure 6: A broken instance detected through the proposed approach in the implementation of secure visitor.

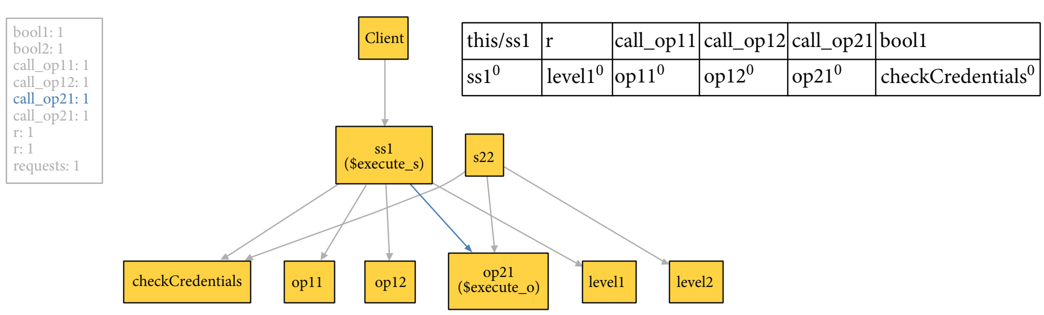


Figure 7: A broken instance detected through the proposed approach in the implementation of the secure strategy factory.

The pattern is rectified by incorporating the predicate for accepting using credentials and creating an unlocked data node that the user will visit. Predicate create and assertion unique is added in the implementation of secure visitor pattern for specifying the constraint. One locked data node will create only one unlocked data node, and unlocked data node should not access are not or child data nodes.

$$\text{pred accept}\,[v : \text{SecureVisitor}, u : \text{UserCredentials}]\{\,\},$$
$$\text{pred create}[l : \text{LockedDataNode}, u : \text{UnLockedDataNode}]\{\,\}. \qquad (2)$$

4.2.3. Secure Strategy Pattern. The class diagram of secure strategy factory is executed to detect any hidden broken access to operations. The secure strategy factory concrete classes are built for two trust levels. TrustLevel-1 can perform op11 and op12, and TrustLevel-2 can operate op21. On executing the predicate *created* for two or more instances in secure strategy factory, an instance is found such that TrustLevel-1 could access the operation op21 restricted for TrustLevel-2. The broken instance is detected using the proposed approach while implementing the secure strategy factory pattern and is shown in Figure 7, without checking user credentials. It is detected through the broken connection that op21 has been inappropriately defined in class meant for TrustLevel-1 and called by TrustLevel-2. The security pattern is rectified by implementing the predicate that $\forall x \in \text{LevelAccess}\ \exists! x\,(f, m)$ such that

$$\exists f \in subclasses(\text{SecureStrategy}) \wedge,$$
$$\exists m \in \text{subclasses}(\text{secureMethod}) \wedge, \qquad (3)$$
$$\text{define}(f, m, x\}.$$

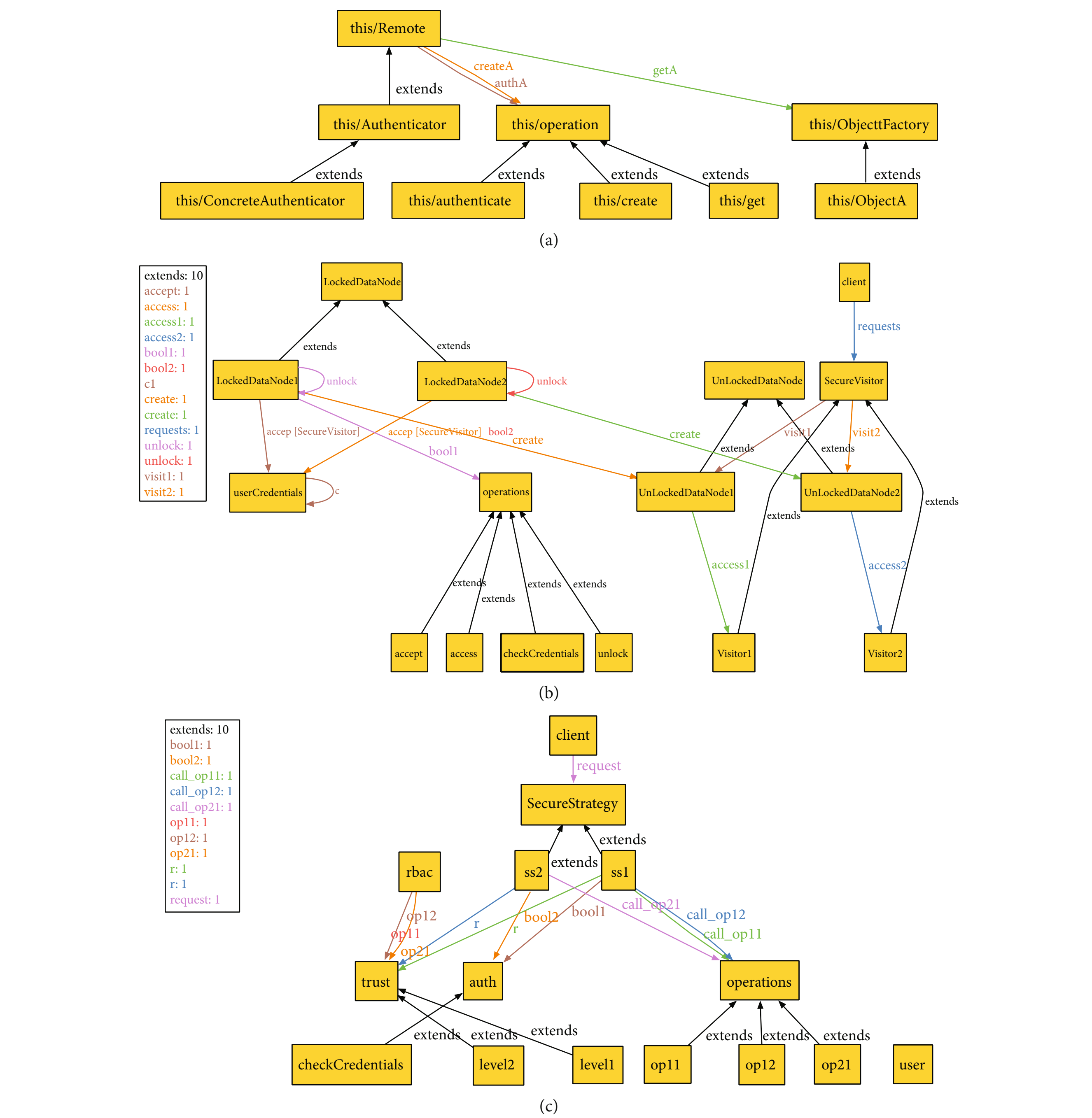


FIGURE 8: The metamodel generated by the proposed approach: (a) authenticator; (b) secure visitor; (c) secure strategy factory.

4.3. Metamodel of Authenticator Pattern, Secure Visitor Pattern, and Secure Strategy Pattern. In this section, the extracted microarchitecture of the security pattern and its rectified predicate logic is executed in the Alloy tool to verify that it meets the security requirement. Each pattern has been executed for multiple instances to create objects, and each instance is analysed for the existence of any counterexample. The metamodel of the authenticator pattern is generated and shown in Figure 8(a). Consider two different authentication systems in a web application. Each of the two authentication systems is implemented using a separate concrete class. The concrete authenticator defines its concrete object and authentication mechanism. The separation of different authentication mechanisms ensures that the other authenticator does not create the object of one authenticator. The arrangement is easily extendible if a third or more authentication mechanisms are appended to the application in future versions. The class diagram of the authenticator pattern is verified using the security constraints written in first-order predicate logic as an assertion *unique*.

In the metamodel of secure visitor pattern shown in Figure 8(b), the client requests SecureVisitor Interface to visit UnlockedDataNode through its concrete implementationsVisitor1 or Visitor2. SecureVisitor can implement any

Table 1: Concrete instances generated by the proposed approach for secure strategy factory.

Concrete instances	Authenticator pattern				Secure visitor				Secure strategy factory			
	No. of variables	No. of primary variables	No. of clauses	Time in ms	No. of variables	No. of primary variables	No. of clauses	Time in ms	No. of variables	No. of primary variables	No. of clauses	Time in ms
2	154	24	225	3	251	44	345	5	1250	285	2177	17
4	432	56	677	3	865	114	1329	8	2694	569	4737	15
6	806	96	1297	5	1699	216	2676	9	4050	861	7150	19
8	1276	144	2085	8	2837	350	4533	13	5618	1161	9920	25
10	1852	200	3121	13	3735	516	6073	15	7288	1469	12938	19
20	6132	600	10701	28	11025	1826	18243	20	16278	3129	29098	27
40	21892	2000	39061	74	42045	6846	70903	73	42178	7049	75878	69
60	47252	4200	85021	113	93065	15066	157963	911	77678	11769	140258	88
128	205824	17664	372933	196	416133	66950	711279	944	270186	33801	490798	270
256	803812	68096	1466821	11196	1651461	264966	2831599	10804	933610	100361	1702574	2125

number of concrete visitors as per the user requirement. LockedDataNode accepts concrete visitor and user credentials and verifies them from UserCredentials class. After checking the user credentials, the data node is unlocked by its respective locked data node and the access is passed to the respective concrete visitor.

In the metamodel of the secure strategy factory pattern shown in Figure 8(c), the client requests SecureStrategy Interface to create factory objects for two different TrustLevel-1 and TrustLevel-2. After checking the user credentials, each role can perform the operations restricted to its role. The class *RBAC* contains policies for TrustLevel and operation mapping. It defines the operations that are restricted to a particular TrustLevel. In the present instance, TrustLevel-1 can perform operations op11 and op12. TrustLevel-2 can perform operation op21. SecureStrategy ss1 creates an object for a user with TrustLevel-1 after checking the credentials. Accordingly, a user with role1 can perform actions op11 and op12 but not operation op21. Similarly, SecureStrategy ss2 creates an object for a user with TrustLevel-2 after checking the credentials. Accordingly, a user with TrustLevel-2 can perform actions op21 but not operations op11 and op12. The security feature of the secure strategy factory is verified using the predicate logic $\forall x \in$ LevelAccess$\exists! x\,(f, m)$ such that

$$\begin{aligned} &\exists f \in \text{subclasses}(\text{SecureStrategy}) \wedge, \\ &\exists m \in \text{subclasses}(\text{secureMethod}) \wedge, \\ &\text{define}(f, m, x\}. \end{aligned} \tag{4}$$

4.4. Validation and Verification of Multiple Concrete Instances. The authors have used the Alloy analyser to verify the proposed approach. The analyser transforms predicate models into Conjunctive Normal Forms (CNF) and analyses using Satisfiability (SAT) solvers. The analyser generates two forms of valuations for the relations in the model: (1) instances, i.e., valuations such that the formulas hold, and (2) counterexamples, i.e., valuations such that the negation of the formulas holds. Using an analyser reduces the time required to verify the model having many literals and clauses in CNF. The microarchitecture of the security pattern extracted through the proposed approach is translated and executed using Alloy. The graphical representation of the metamodel of the security pattern is generated from the extracted microarchitecture and security constraints. The security pattern is executed multiple times for many concrete instances to check the consistency of relationships among the various member elements. Several concrete instances of authenticator, secure visitor, and secure strategy factory are created in the Alloy analyser to validate the model further. Alloy generates the number of concrete instances, primary variables, and clauses in CNF. The simulation time taken to verify each concrete instance for the authenticator, secure visitor, and secure strategy pattern is given in Table 1. No clause with broken access control for accessing the restricted data element has been found in these instances. The correctness of the proposed metamodel for each security pattern is verified.

4.5. Analysis and Discussion. The proposed approach represents the design, structure, dynamics, and other dependencies among various components of security patterns in the generated metamodel. The approach will enable web developers to implement code accurately and minimise threats to the web application. The proposed approach provides a general model that applies to every domain. It considers structural aspects and interrelationships among various structural elements and analyses behaviour based on possible relationships, states, transitions, and execution steps. The proposed approach verifies security patterns by defining the set of interface, abstract, and concrete classes and static methods and nonstatic method calls on the class and identifier and all dependencies at the design phase. It will be helpful in verifying applications. All security-related constraints, artefacts, and static and dynamic checks mentioned at different sections of a security pattern are represented and validated using the predicate modelling before implementation.

It provides precise semantics, verification and validation, and automated reasoning and is machine incomprehensible due to the usage of predicate logic. This specification facilitates automated verification and validation of security patterns applied in web applications. For example, in the secure strategy factory pattern, the constraint for every trust level, i.e., complete, partial, and none, should be a separate unique ConcreteObject implementing each trust level. This constraint is specified in predicate as $\forall x \in \text{TrustLevel} \exists ! y \in$ ConcreteObject. Such representations in predicate logic are easily verified using automated tools and help identify any counterexample(s) simple and error-free.

The proposed specification and its metamodel of security pattern defined various structural elements, their relations, and how these elements will communicate using directed edges and appropriate markers. This form of representation enables developers to understand the complete structure. It helps implement an effective and productive design that is instantly easy to verify, validate, and correct at early stages to ensure security properties of confidentiality, integrity, availability, accountability, nonrepudiation, authentication, and authorisation. The proposed approach provides a verifiable specification of structural design and implementation of security patterns covering all security patterns. Moreover, security constraints such as accessibility among micro architecture elements and parent and child nodes that require extraordinary validation, verification, and testing are also analysed.

In the proposed approach, any flaw in the system's design concerning security functionalities and role-based access to resources is found early through structural and behaviour validations, thereby preventing any consequent security breach.

5. Conclusions and Future Work

Security breach due to broken authentication and access control has been an essential concern for web developers. Though security patterns are applied in a web application, there is a need for a method that can verify the correctness of the class diagram built during the design phase. In this paper, the proposed approach verifies the relationships and accessibility among objects and classes of a security pattern applied in a web application. The approach extracts the structural properties, relations, associations, security-related constraints, artefacts, and static and dynamic checks of the class diagram of a security pattern. The extracted microarchitecture is executed using Alloy to identify unauthorised and broken access in the security pattern applied in a web application. The proposed approach detected inconsistencies at the design phase when applied to secure strategy factory, authenticator pattern, and secure visitor pattern. The detected inconsistencies are then rectified by redefining the methods, classes, relationships, and security constraints and subsequently verifying them using predicate logic. The experiments have generated more than 200 instances of concrete classes each for the secure visitor pattern, authenticator, and secure strategy factory pattern. These instances have been verified for the existence of any counterexample. The final metamodel is generated for each security pattern to develop and implement the code. The proposed approach is helpful for the developer community to verify the consistency of relationships among the various member elements of complex class diagrams restricted to different roles, trust levels, and authentication methods. The complex class diagram of the security pattern can be quickly evaluated and verified for secure access control during the design stage of the web application.

The authors intend to extend the approach to verifying the composition of security patterns in web applications in the future.

Data Availability

The proposed approach has been applied to security patterns: Authenticator, Secure Visitor, and Secure Strategy available at [34, 35].

Conflicts of Interest

The authors declare that they have no conflict of interest.

References

[1] OWASP Foundation, *OWASP Top 10 2021*, 2021, April, 2022, https://owasp.org/Top10/.

[2] PT Analytics Security, *CyberSecurity: Threat Landscape*, 2022, April, 2022, https://www.ptsecurity.com/ww-en/analytics/.

[3] C. Gupta, R. K. Singh, and A. K. Mohapatra, "A survey and classification of XML based attacks on web applications," *Information Security Journal: A Global Perspective*, vol. 29, no. 4, pp. 183–198, 2020.

[4] A. Agrawal and R. K. Singh, "Mining software repositories for revision age-based co-change probability prediction," *International Journal of Open Source Software and Processes (IJOSSP)*, vol. 11, no. 2, pp. 16–32, 2020.

[5] A. Agrawal and R. K. Singh, "Predicting co-change probability in software applications using historical metadata," *IET Software*, vol. 14, no. 7, pp. 739–747, 2020.

[6] A. Agrawal and R. K. Singh, "Ripple effect identification in software applications," *International Journal of Open Source Software and Processes (IJOSSP)*, vol. 11, no. 1, pp. 41–56, 2020.

[7] C. Gupta, R. K. Singh, and A. K. Mohapatra, "Securing web applications using security patterns," in *Proceedings of International Conference on Information and Communication Technology for Competitive Strategies*, Udaipur, India, 2019.

[8] M. Bunke, "Software-security patterns: degree of maturity," in *EuroPLoP 2015: 20th European Conference on Pattern Languages of Programs*, pp. 1–17, Kaufbeuren, Germany, 2015.

[9] M. Bunke, *Security-Pattern Recognition and Validation*, Doctoral dissertation, Universität Bremen, 2018, https://media.suub.uni-bremen.de/bitstream/elib/1571/1/00107086-1.pdf.

[10] J. Dong, T. Peng, and Y. Zhao, "Automated verification of security pattern compositions," *Information and Software Technology*, vol. 52, no. 3, pp. 274–295, 2010.

[11] A. Mourad, H. Otrok, and L. Baajour, "New approach targeting security patterns development and deployment," *Information Security Journal: A Global Perspective*, vol. 20, no. 4-5, pp. 231–244, 2011.

[12] G. Pedroza, L. Apvrille, and D. Knorreck, "AVATAR: a SysML environment for the formal verification of safety and security

properties," in *2011 11th Annual International Conference on New Technologies of Distributed Systems*, pp. 1–10, Paris, France, 2011, May.

[13] T. Heyman, R. Scandariato, and W. Joosen, "Reusable formal models for secure software architectures," in *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pp. 41–50, Helsinki, Finland, 2012, August.

[14] C. Castellanos, T. Vergnaud, E. Borde, T. Derive, and L. Pautet, "Formalisation of design patterns for security and dependability," in *Comparch '13: Federated Events on Component-Based Software Engineering and Software Architecture*, pp. 17–26, Vancouver, British Columbia, Canada, 2013.

[15] P. N. Devyanin, A. V. Khoroshilov, V. V. Kuliamin, A. K. Petrenko, and I. V. Shchepetkov, "Formal verification of OS security model with Alloy and Event-B," in *International Conference on Abstract State Machines*, pp. 309–313, Berlin, Heidelberg, 2014, June.

[16] A. J. Varela-Vaca and R. M. Gasca, "Formalization of security patterns as a means to infer security controls in business processes," *Logic Journal of the IGPL*, vol. 23, no. 1, pp. 57–72, 2015.

[17] B. Hamid, S. Gürgens, and A. Fuchs, "Security patterns modeling and formalization for pattern-based development of secure software systems," *Innovations in Systems and Software Engineering*, vol. 12, no. 2, pp. 109–140, 2016.

[18] A. A. Alzahrani, *Using Codecharts for Formally Modelling and Automating Detection of Patterns with Application to Security Patterns*, Doctoral dissertation, University of Essex, 2016, http://repository.essex.ac.uk/16112/1/Thesis%20Final%20To%20Library.pdf.

[19] A. K. Dwivedi and S. K. Rath, "Formalization of web security patterns," *INFOCOMP Journal of Computer Science*, vol. 14, no. 1, pp. 14–25, 2015.

[20] A. K. Dwivedi, S. K. Rath, and S. L. Chakravarthy, "Formalisation of SOA design patterns using model-based specification technique," in *Proceedings of International Conference on Computational Intelligence and Data Engineering*, pp. 95–101, Singapore, 2019.

[21] D. Jackson, "Alloy," *Communications of the ACM*, vol. 62, no. 9, pp. 66–76, 2019.

[22] X. He and Y. Fu, "Modelling and analyzing security patterns using high-level petri nets," *SEKE*, pp. 623–627, 2016.

[23] J. P. Near and D. Jackson, "Finding security bugs in web applications using a catalogue of access control patterns," in *ICSE '16: 38th International Conference on Software Engineering*, pp. 947–958, Austin, Texas, 2016, May.

[24] A. Behrens, "What are security patterns? A formal model for security and design of software," in *ARES 2018: International Conference on Availability, Reliability and Security*, pp. 1–6, Hamburg, Germany, 2018, August.

[25] A. van Den Berghe, K. Yskout, and W. Joosen, "Security patterns 2.0: toward security patterns based on security building blocks," in *2018 IEEE/ACM 1st International Workshop on Security Awareness from Design to Deployment (SEAD)*, pp. 45–48, Gothenburg, Sweden, 2018, May.

[26] H. Shimamoto, N. Yanai, S. Okamura, J. P. Cruz, S. Ou, and T. Okubo, "Towards further formal foundation of web security: expression of temporal logic in Alloy and its application to a security model with cache," *IEEE Access*, vol. 7, pp. 74941–74960, 2019.

[27] F. Obeid and P. Dhaussy, "Formal verification of security pattern composition: application to SCADA," *Computing and Informatics*, vol. 38, no. 5, pp. 1149–1180, 2019.

[28] H. Gadouche, Z. Farah, and A. Tari, "A valid and correct-by-construction formal specification of RBAC," *International Journal of Information Security and Privacy*, vol. 14, no. 2, pp. 41–61, 2020.

[29] A. Gallon, R. Gallier, and E. Bruno, "Access controls for IoT networks," *SN Computer Science*, vol. 1, no. 1, pp. 1–13, 2020.

[30] C. Gupta, R. K. Singh, and A. K. Mohapatra, "A formal approach for implementing security constraints in security patterns," in *2021 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, pp. 1–7, Noida, India, 2021, September.

[31] C. Gupta, R. K. Singh, and A. K. Mohapatra, "GeneMiner: a classification approach for detection of XSS attacks on web services," *Computational Intelligence and Neuroscience*, vol. 2022, Article ID 3675821, 12 pages, 2022.

[32] I. Bayley and H. Zhu, "Formal specification of the variants and behavioural features of design patterns," *Journal of Systems and Software*, vol. 83, no. 2, pp. 209–221, 2010.

[33] H. Zhu, "On the theoretical foundation of meta-modelling in graphically extended BNF and first order logic," in *2010 4th IEEE international symposium on theoretical aspects of software engineering*, pp. 95–104, Taipei, Taiwan, 2010, August.

[34] E. B. Fernandez, N. Yoshioka, H. Washizaki, and J. Yoder, "Abstract security patterns and the design of secure systems," *Cybersecurity*, vol. 5, no. 1, pp. 1–17, 2022.

[35] C. Dougherty, K. Sayre, R. C. Seacord, D. Svoboda, and K. Togashi, *Secure Design Patterns (No. CMU/SEI-2009-TR-010)*, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 2009.