

Model Checking Security Pattern Compositions

Jing Dong, Tu Peng, Yajing Zhao
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083, USA
{jdong, txp051000, yxz045100}@utdallas.edu

Abstract

Security patterns capture best practice on secure software design and development. They document typical solutions to security problems. To ensure security, large software system design may apply many security patterns to solve different problems. Although each security pattern describes a good design guideline, the compositions of these security patterns may not be consistent and encounter problems and flaws. In this paper, we present an approach to model checking the compositions of security patterns. In this way, the properties of the security patterns can be checked by a model checker when they are composed. Composition errors and problems can be discovered early in the design stage. We also use a case study to illustrate our approach and show the detection of several errors.

KEYWORDS

Design pattern, security, logics, process algebra

1. Introduction

Security is one of the important factors of software quality. With the increasing security attacks, security concern becomes a critical requirement for successful software systems. Studies [35] have shown that design flaws and errors are normally the main source of security holes that are explored by attackers. Therefore, secure software architecture and design are at the heart of software security.

Security patterns [29][31] document good practices to solve security problems arising frequently in software development. A security pattern is a design pattern [18] that generally describes a group of participants and their relationships and collaborations, which achieve some security goals. Each participant in the group is defined generically in terms of the role it plays in the security pattern. A security pattern is a recipe of solving a particular security problem. The benefits of security patterns include the reuse of security design solutions instead of the reuse of just program, document of expert design experience, record of security design tradeoffs,

capture of security decisions, and improvement of communication.

Large software systems normally face many security problems that can be solved by different security patterns. Combining security patterns may help to solve multiple security problems in the same system. While each security pattern describes an expert experience on solving a particular security problem, the composition of these security patterns may not always be good solutions. There can be interactions among security patterns such that some critical security properties may no longer hold. The inconsistencies between security patterns may cause problems in the design. Discovering these problems and errors early in the design is important because such design errors are very difficult to find and correct when they are transformed to implementation errors. Analysis techniques that help to find such design errors are crucial to the quality of the software systems.

Model checking is an automated verification technique that has been initially applied in the hardware community to verify safety and liveness properties [4][7][16]. It has also been used in the software community, e.g., in requirement analysis [3], in distributed cache coherence analysis [36], in hypermedia applications [12], and in Java meta-locking algorithm analysis [5]. In this paper, we use model checking techniques to analyze the consistency of security pattern compositions. More specifically, we formally specify the behavioral aspect of the security patterns, as well as the properties of each security pattern. A model checker is used to perform the analysis and check whether the characteristics of each security pattern still hold after they are composed. Our analysis results show that our approach is able to find the design errors that may lead to security holes and flaws.

The remainder of this paper is organized as follows: the next section describes our analysis techniques on security pattern compositions. Section 3 presents a case study to illustrate our approach and show the discovery of several subtle security problems in the design. The last two sections cover related work and conclusions.

2. Our Security Analysis Techniques

In order to analyze the compositions of design patterns, we apply model checking techniques. Model checking is a method of verifying algorithmically a formula against a logic model [7]. This verification technique can be automated for some temporal logics. In our case, we assume a logic model representing the security patterns and their compositions and a logic formula representing a property of these patterns. In order to establish whether the pattern-based system satisfies the properties, it is checked whether the formula holds in the logic model of security pattern compositions.

There are several model checking tools, such as SMV [22], SPIN [19], and XMC [26]. In this paper, we will concentrate on the XMC model checker since we can use it to analyze both the structural and behavioral aspects of security patterns and their compositions based on Prolog [9]. XMC is a model checker for verifying temporal properties of a system. It is written in the XSB tabled Prolog programming system [39]. Temporal properties are expressed in the alternation-free fragment of the μ -calculus [21][32]; the system to be verified is described in the specification language for XMC (called XL) which is a highly expressive extension of value-passing CCS [25]. Prolog terms and predicates are used to represent values and computations, respectively. In addition, specifications can make use of recursive data structures and computations. The syntax of XL specification is as follows:

```
Pdef --> ( Pname ::= Pexp . ) *
Pname --> Prolog Term
Pexp --> Pexp o Pexp      Prefix
| Pexp # Pexp            Choice
| Pexp '|' Pexp          Parallel Composition
| Pexp @ PortMap         Relabelling
| Pexp \ PortList        Restriction
| Pname                  Recursion
| in(Port,Term)           Communication (input)
| out(Port,Term)          Communication (output)
| Action                  Communication (non-sync)
| Comp                    Computation (Prolog expression)
| if(Comp,Pexp,Pexp)      Conditional Expression
| zero                    Empty process (0 in CCS)
| nil                     Empty computation
PortMap --> [PortTerm/PortTerm (, PortTerm/PortTerm)*]
PortList --> { PortTerm (, PortTerm)* }
Port(Term) --> Prolog Term
Action --> Prolog Atom
```

Pname is a parameterized process name, represented as a Prolog term; Comp is a computation, e.g., X is Y+1; the process in(Port,Term) inputs a value over port Port and unifies it with term Term; out(Port,Term) outputs

term Term over port Port; the process if (Comp, Pexp, Pexp) behaves like the first Pexp if computation Comp succeeds and otherwise like the second Pexp. The operation ‘o’ denotes sequential composition; ‘|’ is parallel composition; ‘#’ is nondeterministic choice; ‘@’ is relabeling where PortMap is a list of substitutions; and ‘\’ is restriction where PortList is a list of port names. Recursion is provided by a set of process definitions, Pdef, of the form Pname ::= Pexp.

Temporal properties are expressed in the modal μ -calculus whose semantics is usually described over sets of states of labeled transition systems. The μ -calculus is encoded in XMC in an equation form as follows:

```
D --> Z += F (least fixed point)
| Z -= F (greatest fixed point)
F --> Z | tt | ff | F ∨ F | F ∧ F | <A> F | [A] F
```

Z is a set of formula variables encoded as Prolog atoms and A is a set of actions; tt and ff are propositional constants; \vee and \wedge are standard logical connectives; $\langle A \rangle F$ denotes that possibly after the action A the formula F holds; $[A] F$ denotes that necessarily after the action A the formula F holds.

Consider, for example, the specification of the Alternating Bit Protocol [34] in XL. We assume that any text after the % character is a comment.

```
medium(Get, Put) ::=
in(Get, Data);
{ out(Put, Data)
# action(drop)
};
medium(Get, Put).

sender(AckIn, DataOut, Seq) ::=
% Seq is the sequence number of
% the next frame to be sent
out(DataOut, Seq);
{
in(AckIn, AckSeq);
if AckSeq == Seq
%% successful ack, next message
then {
NSeq is 1-Seq;
sendnew(AckIn, DataOut, NSeq)
}
%% unexpected ack, resend message
else sender(AckIn, DataOut, Seq)
#
%% upon timeout, resend message
sender(AckIn, DataOut, Seq)
}.

sendnew(AckIn, DataOut, Seq) ::=
action(sendnew);
sender(AckIn, DataOut, Seq).

receiver(DataIn, AckOut, Seq) ::=
```

```

%% Seq is the expected next sequence number
in(DataIn, RecSeq);
if RecSeq == Seq
  then {
    NSeq is 1-Seq;
    action(recv);
    out(AckOut, RecSeq);
    receiver(DataIn, AckOut, NSeq)
  }
  else {
    %% unexpected seq, resend ack
    out(AckOut, RecSeq);
    receiver(DataIn, AckOut, Seq)
  }
}.
abp ::=
  sendnew(R2S_out, S2R_in, 0)
| medium(S2R_in, S2R_out)      % sender -> receiver
| medium(R2S_in, R2S_out)      % receiver -> sender
| receiver(S2R_out, R2S_in, 0).

```

The process `medium` represents a noisy channel. The sender process sends a packet to the channel and waits for an acknowledgement. Upon timeout, it resends the packet. The receiver process receives a packet from the channel and sends an acknowledgement back. The `abp` process is the parallel composition of the previously described processes. Some temporal properties, such as deadlock and drop packet, are described in μ -calculus as follows. These properties can be checked against the model of the Alternating Bit Protocol by XMC.

```

%% A packet can be lost without being received
drop_packet += <sendnew>lost  $\vee$  <->drop_packet.
lost      += <sendnew>tt  $\vee$  <-rcv>lost.

```

```

%% The system can deadlock.
deadlock += [-] ff  $\vee$  <-> deadlock.

```

Figure 1 illustrates the main characteristics of our approach to analyzing security pattern compositions. Initially, each security pattern is formally specified in a declarative way using XL. The security pattern specifications are generic in the sense that they capture good design practice in a domain-independent way. These declarative representations, which constitute models of the security patterns, are then instantiated into concrete domain-specific representations and, in this way, security design practice can be reused. The instances of security patterns are integrated to form a model Σ of the composition of the security patterns, which is then submitted to a model checker. We use the model checker to check Σ against the property specification Φ . The model checker outputs either true, if Σ satisfies Φ , or a counterexample, if it does not.

More specifically, we provide the following guidance to formalizing security patterns in XL. Each

object in the security pattern is modeled as an XL process that inputs some messages, performs actions, and then outputs some messages. Thus, each object is normally specified as an XL process in the following way:

$P ::= \text{in}() \circ \text{action}() \circ \text{out}()$.

where `in()` specifies that the process `P` receives some message from other process. Since `P` corresponds to an object, it means some operation of the corresponding object has been invoked by some other object. A process `P` may have multiple `in()` which specifies that some operations of the object corresponding to `P` have been invoked by other objects. The `action()` specifies that the process perform some action after receiving the message. The action typically changes the internal states of the corresponding process. The process can perform multiple actions as well. Model checking is generally applied on these state changes to verify whether a sequence of state changes is desired. As a consequence, a process may send out some message after performing the actions. The `out()` specifies that the corresponding object of `P` may invoke the operations of some other objects. Similarly, each process may send out multiple messages.

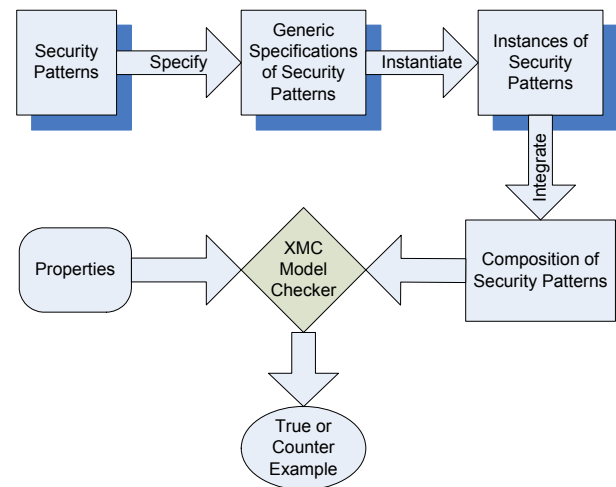


Figure 1 Overview of Our Approach

3. Case Study

In this section, we present a case study to illustrate our approach. In particular, we show the detection of several subtle design errors based on our analysis techniques. More specifically, Section 3.1 describes the Secure Pipe and Authentication Enforcer patterns. Section 3.2 formally specifies these patterns in XL. Section 3.3 introduces an application on secure

observer. Section 3.4 presents our analysis of these security pattern compositions.

3.1 Security Patterns

With the increasing security concerns on software systems, security requirements become more and more important in software design and development. Security is one of the important quality factors of software systems. A security pattern is a well-understood solution to a recurring information security problem, it encapsulates security expertise in the form of worked solutions to these recurring problems.

Many security patterns have been discovered to solve the recurring security problems, such as authentication, authorization, confidentiality, and communication. In this section, we describe two security patterns: the Secure Pipe pattern and the Authentication Enforcer pattern.

One of the common security problems is secure communication between two parties. Unsecured communications are often exposed to eavesdropping, spoofing, sniffer, and replay attacks. The replay attacks copy the legitimate transactions and resend them. The sniffer attacks just capture sensitive information for use later. Many of these attacks can be categorized as man-in-the-middle attacks which can not only harm the unsecured network but also VPN where data is exposed at the end points. This exposed data is still subjected to disclosure, modification, or duplication. Some of these attacks are easy to carry out, even for novices. As a consequence, these attacks may result in huge losses for business that need to communicate sensitive data.

The Secure Pipe pattern provides a solution to the problem and guarantee the integrity and privacy of data sent over the network. The secure pipe does not require application-layer logic and provides a simple and standard way to protect data. The task of securing a pipe is moved to the hardware platform to reduce the complexity of implementation. In some case, it is even moved out of the hardware platform.

Figure 2 shows a class diagram of the Security Pipe pattern related to an application. Initially, the client may login the application. The application then creates a SecurePipe at the system level. The SecurePipe is an encrypted communication channel over which the client communicates with the application. The channel is secure and provides data privacy and integrity between two endpoints. When the client logs out, the application destroys the SecurePipe.

Figure 3 shows a sequence diagram of the Security Pipe pattern which depicts the activities of each object in an application. When the client wants to send information to the application in a secured channel, it will let the application know by calling the “login”

method of the application that creates a securepipe. The client will then call the “request” method to pass the information to the securepipe which encrypts the information and sends to the application.

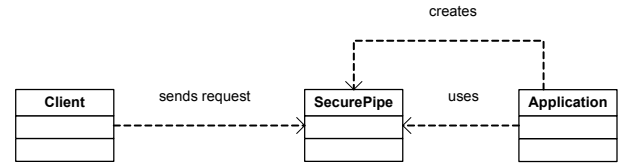


Figure 2 Secure Pipe Pattern Class Diagram

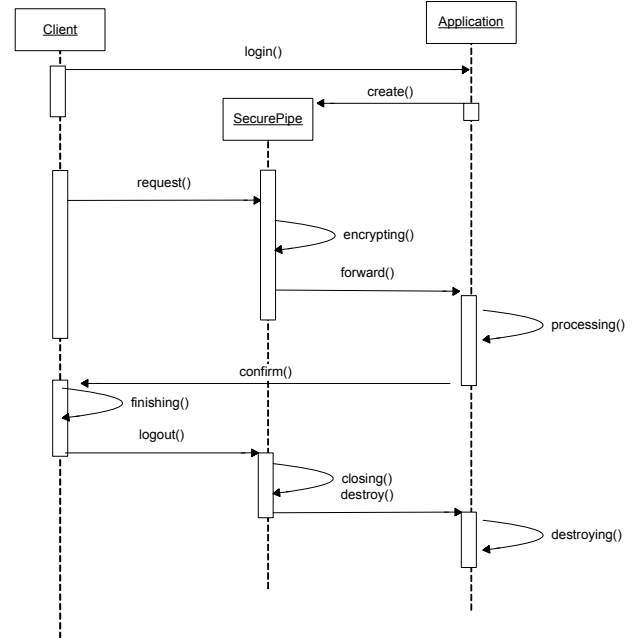


Figure 3 Secure Pipe Pattern Sequence Diagram

Another common security problem is authentication. Only valid users can access certain application resource. These users must be properly authenticated. There are several different ways to authenticate users, such as HTTP basic authentication, form-based authentication, certificate-based authentication, and custom authentication via JAAS. These authentication mechanisms can be used jointly in the same application. Due to changes of business requirements, application-specific characteristics, and underlying security infrastructure, users may change the choice of these authentication mechanisms. In addition, there may be multiple entry points into an application, each requiring user authentication. Therefore, the code related to authentication is duplicated in many places making the system difficult to develop and easy to make mistakes.

The authentication enforcer pattern provides a solution to centralize the user authentication processes and encapsulate the details of the authentication

mechanism. The authentication logic for verifying user identity is delegated to a helper class that interacts with the security providers. This centralized mechanism applies to all different kinds of authentications, such as password-based and client certificate-based. The benefits of centralizing and encapsulating authentication mechanics behind a common interface include eases of authentication requirement evolution and facilitating reuse. The generic interface allows the authentication independent from the protocols.

Figure 4 depicts the Authentication Enforcer pattern where the AuthenticationEnforcer centralizes and encapsulates the authentication logic. When two clients need to authenticate one another, they can achieve that through the AuthenticationEnforcer class. More specifically, the AuthenticationEnforcer class authenticates a user by using the credential passed in the RequestContext that contains the user's credentials. The AuthenticationEnforcer creates a subject instance that represents the authenticated user.

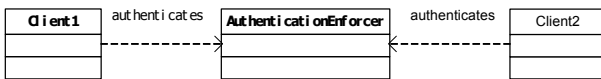


Figure 4 Authentication Enforcer Pattern Class Diagram

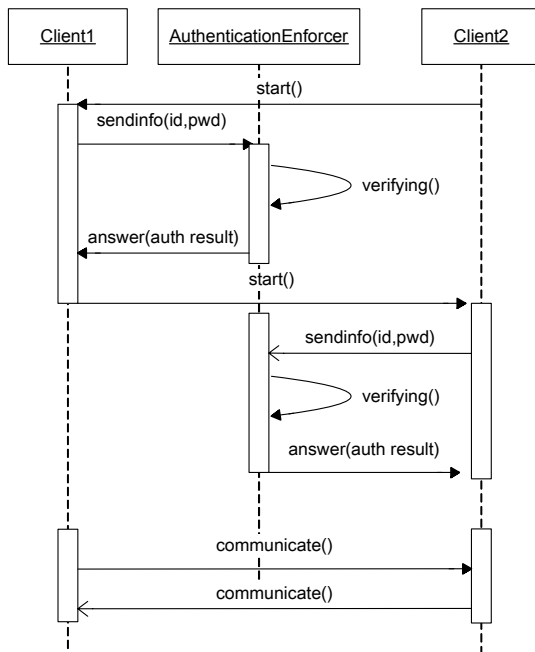


Figure 5 Authentication Enforcer Pattern Sequence Diagram

Figure 5 shows a sequence diagram of the Authentication Enforcer pattern. When client2 wants to build a trust relationship with client1, it calls the “start”

method of client1 that sends its privacy information to AuthenticationEnforcer. The identity of client1 is verified by the AuthenticationEnforcer. After client1 is identified, it calls the “start” method of client2 that will proceed the same way to get itself identified. Since both client1 and client2 are identified by a trustworthy third party, AuthenticationEnforcer, they can then trust each other.

3.2 Formal Specifications

As the applications of security patterns are very sensitive and important, the users want to have confidence that the applications' functions correctly observe their expected behavior. Formal methods can provide rigorous framework and trustable procedure to meet such needs. As discussed previously, we formally specify the security pattern in XL that is based on CCS. In particular, the behavior of each security pattern is formalized as a group of processes and their communications based on our approach in Section 2. The behavioral specification of the Secure Pipe pattern is presented as follows¹:

```

BC(securepipe)::= client | securepipe | application \ {login,
    negotiate, request, forward, confirm, create}
client::= in(start) o action(startup) o out(login) #
    in(negotiate) o action(negotiating) o out(request) #
    in(confirm) o action(finish) o out(logout).
securepipe::=in(request) o action(encrypt) o out(forward) #
    in(create) o action(prepare) # in(logout) o
    action(closing) o out(close).
application::=in(login) o action(creating) o out(create) #
    in(forward) o action(processing) o out(confirm) #
    in(close) o action(destroy).

```

where the Secure Pipe pattern is specified by the parallel composition of three processes: client, securepipe, and application. The client process will send out the login request when it starts. When it gets the negotiation message, it will send out a request to the securepipe process. When the client gets the confirmation, it will logout. When the securepipe process gets the request from the client, it will perform an encryption action and forward the request to the application. When the securepipe process gets the message for creation, it will prepare. The application process will create a secure pipe when it gets login request. When the application gets the request forwarded by the securepipe, it will process the request and send out the confirmation to the

¹ Note that we simplify the process expressions $in(X, X)$ and $out(X, X)$ to $in(X)$ and $out(X)$, respectively, when the port name and message name are the same. The sequential composition ('o') has higher priority than the nondeterministic choice ('#') in XL.

client. When the application receives the request to close, it will destroy the connection.

The behavior of the Authentication enforcer pattern is specified in CCS as follows:

```
BC(authenticationenforcer)::=client1 | authenticationenforcer |
  client2 \ {identity, password, authresult}
client1::=in(start) o action(startup) o out(identity) o
  out(password) # in(authresult) o action(communicate);
authenticationenforcer::=in(identity) o in(password) o
  action(verify) o out(authresult);
client2::=in(start) o action(startup) o out(identity) o
  out(password) # in(authresult) o action(communicate).
```

where the client process will send its identity and password to the authenticationenforcer process when it starts. When the client gets the authentication result, it will start the communication. When the authenticationenforcer process receives the identity and password of a client, it will verify and then send out the authentication result.

3.3 Secure Observer

Let us consider a case of secure observer. The Observer pattern has been introduced in [18], which addresses the problem of maintaining consistency among different copies of the same data. In practice, a data may have different views which present it in some special ways. When the data is changed, all its views need to be notified and changed. The Observer pattern provides a solution to this problem by loosely connecting the data and its views through attaching and detaching mechanism. Figure 6 shows the class diagram of the Observer pattern.

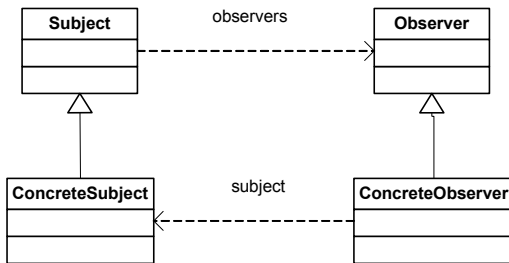


Figure 6 The Observer Pattern Class Diagram

In normal Observer pattern, there is no security issue. However, there are many situations that both the subject and the observers need to authenticate each other. The subject can be viewed only by authorized observers, whereas an observer may only intend to view trusted subject. For example, the sales information of a company can be only viewed by the sales manager who has the most up-to-date information. A stock market watcher may only observe the market information from

trusted source (subject). In addition, the communication between the subject and the observers should be secure. The sensitive data should not be intercepted by malicious attackers. In these situations, we need to add security solutions to the Observer pattern. In particular, the Secure Pipe and Authentication Enforcer patterns can be applied to solve these problems. Figure 7 depicts a composition of the Observer, Secure Pipe, and Authentication Enforcer patterns. The AuthenticationEnforcer class is used to authenticate the subject and observers. The SecurePipe class is used to provide secure communication between the subject and observers.

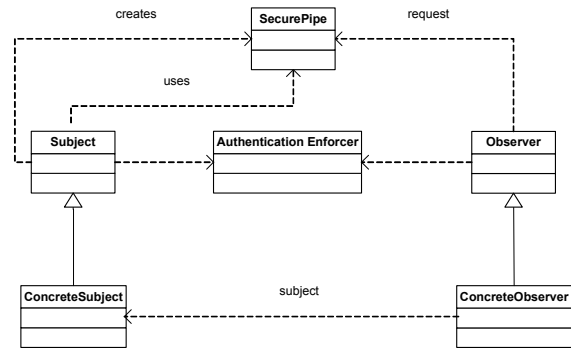


Figure 7 Secure Observer Class Diagram

3.4 Analysis of Security Pattern Compositions

In the previous section, we introduced a composition of two security pattern with the Observer pattern. In this section, we will analyze this composition. We are interested in knowing whether there are any problems or errors in the composition shown in Figure 7.

Based on our analysis approach described in Section 2, we have formally specified the behavioral of the Secure Pipe and Authentication Enforcer patterns in the previous section. The behavioral of the Observer pattern can be specified in XL as follows:

```
BC(observer)::=subject | observer \ {setstate, update}
subject::=in(setstate) o action(changestate) o out(update).
observer::=in(change) o action(change) o out(setstate) #
  in(update) o action(getstate).
```

Therefore, the behavioral composition of the three patterns in Figure 7 can be specified by the parallel composition of the behavioral specification of each pattern as follows:

```
BC(integration)::=BC(observer) | BC(securepipe) |
  BC(authenticationenforcer)
```

In our analysis, we are interested whether the communication between the Subject and the Observers is really secure in this composition. In particular, we are interested in the following properties:

1. The authentication between the Subject and Observer always happens after a secure pipe has been established between them.
2. Any message from Subject to Observer is always sent after they authenticate each other.
3. Any message from Observer to Subject is always sent after they authenticate each other.
4. After authentication, any message sent from Subject to Observer should always be secure.
5. When the Subject and Observer synchronize their internal states, the Observer needs to get the state information from a secure communication channel.

We used the XMC model checker to check these properties and found that the communication is actually not secure. When the Observer tries to authenticate itself to the Subject, it connects to the Authentication Enforcer and provides its identity information to the Authentication Enforcer. However, the problem is that the communications between the Subject/Observer and Authentication Enforcer are not secure. They are not protected by the Secure Pipe. To secure this communication, thus, new Secure Pipes are needed in between the Subject/Observer and the Authentication Enforcer. The improved integration is shown in Figure 8, where two new Secure Pipes, SecurePipeA and SecurePipeB, are added. The behavioral composition of these five patterns is specified in XL as follows:

```
BC(integration)::=(BC(observer) | BC(securepipe) |
  BC(authenticationenforcer) | BC(securepipeA) |
  BC(securepipeB)) @ f
```

where f is a relabeling function that changes a message name to a new one. For example, a relabeling function $g=\{A, B\}$ replaces message A by B . The relabeling function is applied to all messages in the corresponding process. For instance, $P @ g$ replaces all messages A by B in process P . In this case study, f is defined as follows:

```
BC(securepipeA) @ f = { (login,alogin), (create,acreate),
  (negotiate,anegotiate), (request,arequest),
  (forward,afoward), (confirm,aconfirm) }
BC(securepipeB) @ f = { (login,blogin), (create,bcreate),
  (negotiate,bnegotiate), (request,brequest),
  (forward,bforward), (confirm,bconfirm) }.
```

where the relabeling function f on $BC(securepipeA)$ distinguishes the communication between the Subject and the SecurePipe and the communication between the Subject and the SecurePipeA. Similarly, the relabeling

function f on $BC(securepipeB)$ distinguishes the communication between the Observer and the SecurePipe and the communication between the Observer and the SecurePipeB.

Figure 9 shows how the observer verifies its identity by the authentication enforcer through a secured channel. The observer will send its identity information to the securepipe using the “request” method. The securepipe will encrypt this information and forward it to the authentication enforcer, which will decrypt and verify it. The interactions between authentication enforcer and subject are the same.

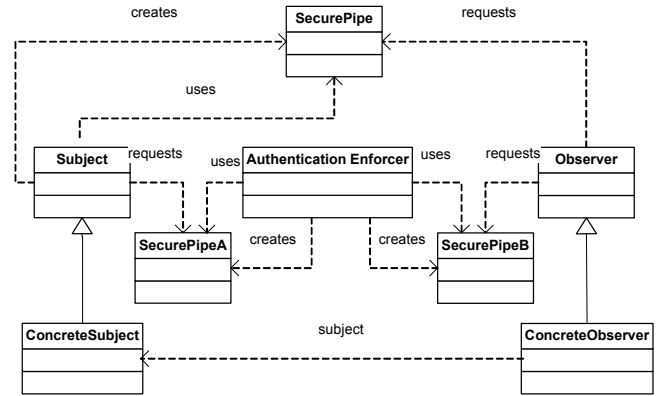


Figure 8 Secure Observer Class Diagram (Revised Version One)

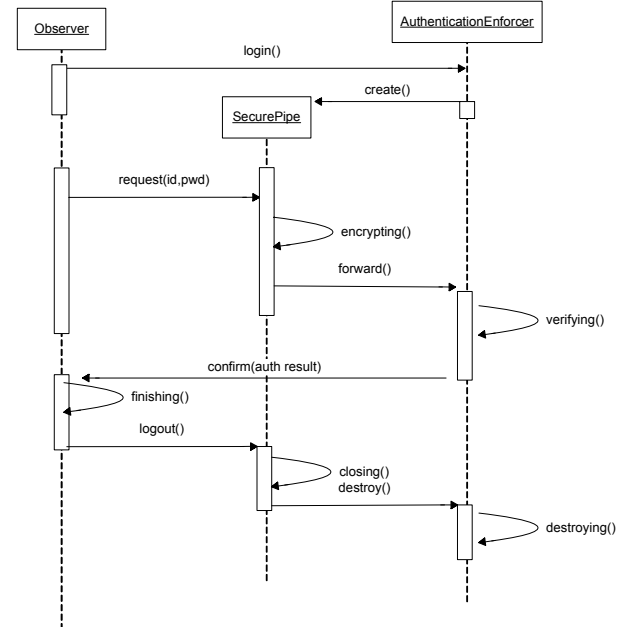


Figure 9 Secure Observer Sequence Diagram (Revised Version One)

formally specified in [30]. Following the ideas of the design by contract approach in [23], the structural and behavioral specifications are captured as responsibilities, whereas the rewards capture the benefits of applying the pattern with the expected behavior in a system.

The composition of two design patterns based on a specification language (DisCo) has been discussed in [24]. The behavior of each pattern is formalized as a layer in DisCo. The composition of design patterns is defined as a refinement on the layers of specifications.

Formal specification of design patterns and their composition based on the Language of Temporal Ordering Specification (LOTOS) is proposed in [28]. In particular, the behavioral aspect of the Command and Composite patterns and their combination is specified.

Property patterns [14] have been proposed to provide taxonomy of properties written in LTL, CTL and QRE. Each property pattern is defined in terms of its scope, which is the extent of the program execution over which the pattern should hold. Property patterns define general properties whereas we focus on security-related properties.

5. Conclusions

Security patterns have been adopted in software industry to reuse expert design experience on solving security problems. While each security pattern presents a good solution to a security problem, there lacks work on analyzing their compositions that may have inconsistencies and interactions. Failure of detecting such errors and problems may result in security holes that suffer malicious attacks. Analysis at high level (architecture and design) may greatly save time and effort than that at lower implementation level.

In this paper, we provide an approach to analyzing the compositions of security patterns using model checking techniques. In our approach, we presented the guideline to specify the behavior of security pattern in the model specification language. We also conducted a case study to show the detection of several security problems. Our results showed that our approach can find several subtle errors in the compositions.

References

- [1] Paulo Alencar, Donald Cowan, Jing Dong, and Carlos Lucena, A Pattern-Based Approach to Structural Design Composition, *Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference*, pages 160-165, Phoenix USA, October 1999.
- [2] Paulo Alencar, Donald Cowan, and Carlos Lucena. A Formal Approach to Architectural Design Patterns. *Proceedings of the Third International Symposium of Formal Methods Europe (FME)*, pages 576–594, 1996.
- [3] Joanne M. Atlee and John Gannon. State-Based Model Checking of Event-Driven System Requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.
- [4] Geoff Barrett. Model Checking in Practice: The T9000 Virtual Channel Processor. *IEEE Transactions on Software Engineering*, 21(2):69–78, November 1998.
- [5] Samik Basu, Scott A. Smolka, and Orson R. Ward. Model Checking the Java Meta-Locking Algorithm. *Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS)*, pages 342–350, April 2000.
- [6] F. L. Brown and E. B. Fernandez, The authenticator pattern. *Proceedings of the Pattern Languages of Programs (PLOP'99)*, 1999.
- [7] M.C. Browne, Edmund M. Clarke, and D.L. Dill. Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Transactions on Computer*, C-35(12):1035–1044, Dec. 1986.
- [8] S. Chinnasamy, R. R. Rajee, and Z. Liu. Specification of design patterns: An analysis. *Proceedings of the 7th International Conference on Advanced Computing and Communications*, pages 300–304, 1999.
- [9] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Berlin: Springer Verlag, 1987.
- [10] Jing Dong, Paulo Alencar, and Donald Cowan, Ensuring Structure and Behavior Correctness in Design Composition, *Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS)*, pp279-287, Edinburgh UK, 2000.
- [11] Jing Dong, Paulo Alencar, and Donald Cowan. A Behavioral Analysis and Verification Approach to Pattern-Based Design Composition. *International Journal of Software and Systems Modeling*, Springer Verlag, 3(4):262–272, December 2004.
- [12] Jing Dong, Paulo Alencar, and Donald Cowan. Automating the Analysis of Design Component Contracts. *Software – Practice and Experience (SPE)*, Wiley, 36(1):27–71, January 2006.
- [13] Jing Dong, Tu Peng and Zongyan Qiu, Commutability of Design Pattern Instantiation and Integration, *the Proceedings of the First IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE)*, China, June 2007.
- [14] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. *Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA*, May 1999.
- [15] A.H. Eden and Y. Hirshfeld. Principles in formal specification of object-oriented architectures. *Proceedings of the 11th CASCON, Toronto, Canada*, November 2001.
- [16] E. Emerson and Edmund M. Clarke. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2:241–266, 1982.
- [17] E. B. Fernandez, J. C. Pelaez, and M. M. Larrondo-Petrie. Security Patterns for Voice over IP Networks.

- Proceedings of the International Multi-Conference on Computing in the Global Information Technology (ICCGI'07), 2007.
- [18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [19] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279-295, May 1997.
- [20] D. M. Kienzle, M. C. Elder, D. Tyree, and J. Edwards-Hewitt. *Security Patterns Repository*, 2002.
- [21] Dexter Kozen. Results on the Propositional μ -calculus. *Theoretical Computer Science*, 27:333–354, 1983.
- [22] M. L. McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. *PhD Thesis, Carnegie Mellon University, CMU-CS-92-131*, 1992.
- [23] Bertrand Meyer. Applying “design by contract”. *IEEE Computer*, pages 40–51, October 1992.
- [24] Tommi Mikkonen. Formalizing Design Pattern. *Proceedings of the 20th International Conference on Software Engineering*, pages 115–124, 1998.
- [25] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
- [26] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient Model Checking Using Tabled Resolution. *Proceedings of the 9th International Conference on Computer Aided Verification (CAV), Haifa Israel, LNCS1243, Springer Verlag*, pages 143–154, July 1997.
- [27] D. G. Rosado, C. Gutierrez, E. Fernandez-Medina, and M. Piattini. A Study of Security Architectural Patterns. *Proceedings of the First International Conference on Availability, Reliability and Security (ARES'06)*, 2006.
- [28] Motoshi Saeki. Behavioral specification of GoF design patterns with LOTOS. *Proceedings of the Seventh Asia-Pacific Software Engineering Conference (APSEC)*, pages 408–415, Dec. 2000.
- [29] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, Peter Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. Wiley, 2006.
- [30] Neelam Soundarajan and Jason O. Hallstrom. Responsibilities and Rewards: Specifying Design Patterns. *Proceedings of the 26th International Conference on Software Engineering*, pages 666–675, May 2004.
- [31] Christopher Steel, Ramesh Nagappan, Ray Lai. *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall PTR, 2005.
- [32] Colin Stirling. An Introduction to Modal and Temporal Logics for CCS. *Lecture Notes in Computer Science 491, Springer Verlag*, pages 1–20, 1991.
- [33] Toufik Taibi and David C. L. Ngo. Formal specification of design pattern combination using BPSL. *International Journal of Information and Software Technology (IST), Elsevier-Science*, 45(3):157–170, March 2003.
- [34] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.
- [35] John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2001.
- [36] Jeannette M. Wing and Mandana Vaziri-Farahani. A Case Study in Model Checking Software Systems. *Science of Computer Programming*, 28:273–299, 1996.
- [37] Sherif. M. Yacoub, Hany H. Ammar. *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems*. Addison-Wesley Professional, 2004.
- [38] N. Yoshioka, S. Honiden, and A. Finkelstein. Security Patterns: A Method for Constructing Secure and Efficiently Inter-Company Coordination Systems. *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC'04)*, 2004.
- [39] XSB. The XSB Logic Programming System, Version 2.1. Available from http://www.cs.sunysb.edu/_sbprolog/xsb-page.html, 1999.