Contract-based design patterns: A Design by Contract Approach to Specify Security Patterns

Caine Silva
Lab-STICC, ENSTA Bretagne
Brest, France
caine.silva@ensta-bretagne.org

Sylvain Guérin
Lab-STICC, ENSTA Bretagne
Brest, France
sylvain.guerin@ensta-bretagne.fr

Raúl Mazo
Lab-STICC, ENSTA Bretagne
Brest, France
GIDITIC, Universidad EAFIT
Medellin, Colombia

Joel Champeau
Lab-STICC, ENSTA Bretagne
Brest, France
joel.champeau@ensta-bretagne.fr

ABSTRACT

With the ever growing digitization of activities, software systems are getting more and more complex. They must comply with new usages, varied needs, and are permanently exposed to new security vulnerabilities.

Security concerns must be addressed throughout the entire development process and in particular through appropriate architectural choices. The security patterns are the founding principles to provide the architectural and design guidelines. Nevertheless, researchers have pointed out the need for further research investigations to improve quality and effectiveness of security patterns.

In this paper, we focus on enhancing security patterns specification to improve the security of the systems using them.

Thus, to reach this goal, we present a formal Design by Contract approach to improve the behavioral definition of the security patterns. This approach seeks to define both functional behavior and implicit parts of security design patterns. Our approach includes the contract formalization of security patterns and a comparative implementation on two Java annotation frameworks. The application of the proposal in a proof of concept case highlights the security enforcement at design time or on a legacy source code.

KEYWORDS

Security by design, design by contract, security patterns

ACM Reference Format:
Caine Silva, Sylvain Guérin, Raúl Mazo, and Joel Champeau. 2020. Contract-based design patterns: A Design by Contract Approach to Specify Security Patterns. In *The 15th International Conference on Availability, Reliability and Security (ARES 2020), August 25–28, 2020, Virtual Event, Ireland*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3407023.3409185

ARES 2020, August 25–28, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-8833-7/20/08…$15.00
https://doi.org/10.1145/3407023.3409185

1 INTRODUCTION

There is no single week without an announcement indicating that a system was attacked, and this trend is not likely to stop in the short term. The relative novelty of the software security field along with the lack of real methodology defining how to design secure systems does not help.

To cope with these threats, researchers have identified generic solutions to mitigate system vulnerabilities: authentication, permission handling, resource access, etc. These solutions, known as security design patterns, are meant to provide architectural and functional guidelines to address security in early design stages. Security design patterns, studied in many research papers [7, 20, 22], capture security expertise and can be used throughout the entire system development process (design phase, implementation, test, etc.).

Software system development is now based on existing components in a goal to reuse, to extend or mainly to compose with new components. Mitigating the security risks of such resulting code is a real challenge [16]. Such challenge can be faced through the extensive use of security patterns. Improvement of legacy code using security design patterns however remains problematic [20].

Applying security patterns to legacy code raises two issues: (i) improving pattern definitions to focus on security properties (behavior) rather than architectural constraints and (ii) ensuring that the security properties of the considered pattern are preserved when applied on a composition of existing code.

To tackle these issues, the contribution of this paper is focused on improving security pattern definitions through behavior formalization. This formalization is based on the design-by-contract approach, presented in Section 2. Section 3 provides a formal definition of contract-based security patterns. We show how this approach can be implemented in two different annotation-based languages in Section 4. Section 4.3 presents the results of both experiments and reports the benefits and limitations that we found in our approach when applied to secure legacy code. To the best of our knowledge, there is no research specifically focused on the extension of security patterns with the notion of formal contracts in order to create well defined security properties in the form of cyber-security contracts. One of the implementations of our approach shows its capacity to

improve the security either of a software component at design time, or of a composition of legacy code, or both.

2 PRELIMINARIES

In order to ensure the security of a software system, not only is it important to design a robust security architecture (intended) but it also is necessary to preserve the security of the (implemented) architecture during software evolution as presented in [1]. Therefore, security knowledge is usually presented as reusable techniques to help engineers and developers designing, implementing and maintaining secure systems. Some of these approaches are, for instance, defensive programming, design security patterns and design by contracts. This section introduces the last two approaches as they constitute the basis of the contract-based security pattern approach presented in this paper.

2.1 Security design patterns

In software engineering, a design pattern is a general, reusable solution to a commonly occurring problem within a given software design context. It is not a finished design that can be transformed directly into source or machine code. It rather is a description or template for how to solve a problem that can be used in many different situations.

A design pattern is classically composed of three parts:

- The *problem*. This part describes in a few sentences which problem is addressed by the pattern.
- The *solution*. This part is usually composed of a UML (Unified Modeling Language) class diagram and a few sentences to explain the diagram.
- The *remarks*. The authors of the pattern can add in this part any information they think is relevant. It usually involves performance results.

With the rise of cyber security, new design patterns – security design patterns – have been developed to provide generic solutions to fulfill some common security goals such as authentication, authorization and secure message delivery. The following subsection presents the authentication pattern. It is also used throughout this paper to exemplify our approach and implementations.

2.1.1 Authentication pattern. The goal of this pattern is to verify the identity of a subject. Such a subject can be someone interacting with a software by providing credentials, a subsystem requesting data to another subsystem, a thread requesting extra privilege to an operating system, etc. The pattern only provides an architecture which is compatible with the use of every authentication algorithm. The architecture of the Authentication pattern is presented in Figure 1.

It is composed of four parts:

- a *Subject* needing to be authenticated,
- a *Proof of Identity*, token given to the subject once the authentication is complete,
- an *Authenticator* is the object which implements an authentication algorithm and creates the *Proof of Identity*,
- *Authentication Information* are the information provided by the *Subject* to the *Authenticator*.

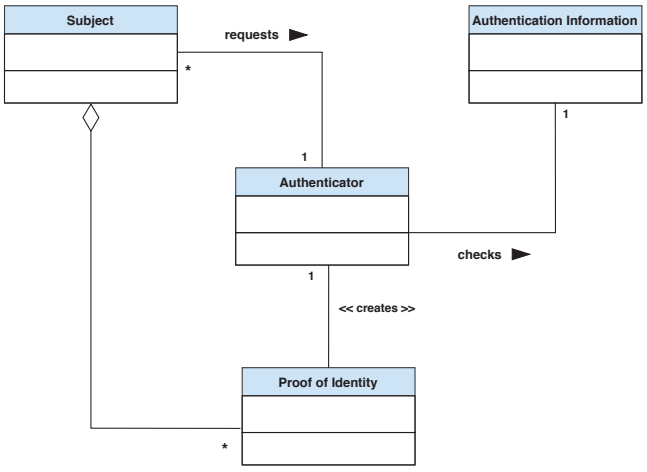

Figure 1: UML class diagram of the Authentication pattern

2.1.2 Limitations of design patterns. The use of design patterns has many advantages for software development as they bring a certain uniformity in software design. They provide a common language between software designers and developers and make software more easily maintainable and extensible. As a generic solution, the definition of a design pattern is usually abstracted from the implementation. Although this abstraction allows for pattern reusability, it is problematic when it comes to security. Indeed, most attacks on software systems are based on weaknesses in the implementation, referred to as vulnerabilities. The presence of natural language in the definition of patterns also introduces potential vulnerabilities, as it is subject to interpretation. In particular, the implementation details are usually implied in the security pattern definition. Therefore, the use of security design patterns does not guarantee the security of a software, as their definition is not formal enough, and thus ambiguous.

2.2 Design by contract

The Design by Contract (DbC) concept was coined by Meyer [14] as an approach to design reliable software based on the idea that elements of a software system collaborate with each other on the basis of mutual obligations and benefits.

The whole DbC approach relies on the idea of contract. Meyer indeed realized that most of the software systems, and in particular object-oriented systems, depend on the division of work. This means that tasks are classically divided in several sub-tasks, each being conducted by a program unit. Most of the time, the completion of a given task is made possible by the division of labor between several actors. When this happens, the actual interaction of the actors is entirely defined in a *contract*. This contract contains the liabilities and benefits of the interaction for all parties involved. This analogy led Meyer to the idea of software contracting [14]: to define contracts between clients (i.e., routine's callers) and suppliers (i.e., routines, functions or methods). A contract is defined as the aggregation of two assertions to a routine or method:

- A precondition: Boolean condition that needs to be verified before calling the routine. It summarizes the client's obligations towards the supplier.
- A postcondition: Boolean condition that needs to be verified after the call is made. It summarizes the supplier's obligations towards the client.

The whole idea of the DbC approach is that since a contract is formally defined for each service (routine or method), bugs are unlikely to appear at run-time because of a misunderstanding between program units.

In addition to these two assertions, Meyer defined a third type of Boolean expression called a *class invariant*. This notion relies on the class concept. In object-oriented designs, a class should be the representation of some specific concept, usually referred to as object or model. Most of the time, a few properties characterize the essence of the class and should be true at all time and for every instance. A classic example of this idea is the binary tree node class in which all nodes are connected to at most two nodes. A node instance of such a structure should verify at any time that both its children reference it as their respective parent node. This property is then an *Invariant* for this class.

3 USING CYBER CONTRACTS TO SPECIFY SECURITY PATTERNS

The design-by-contract approach [14] states that faults in a software system are the consequence of inadequate programming practices, in contrast to other schools of thought that impute the security problem to even earlier stages of system definition such as requirements elicitation [17] [13] [12]and modelling [18]. However, this approach was not designed to deal with security issues. Thus, the notion of cyber contract presented in this section, is an extension of the DbC method with concepts coming from the cyber security domain. In particular, this section (i) formally describes cyber security contracts, (ii) presents the application of such contracts to security patterns, and (iii) develops the case of the authenticator pattern to illustrate our approach.

3.1 Cyber security contracts

The core construct of a cyber security contract is the concept of *contract*, which links formal properties and contracting parties as depicted in the metamodel presented in Figure 2. The class *Contract* has an attribute *intent*, has a collection of contracting parties and is composed of *Clauses*. The *intent* is a text that describes, in natural language, the purpose of the contract. The contracting parties are represented by the *ContractingParty* class and they correspond to the programming entities whose behaviors are subject to the *Contract*. These entities can be classes, methods, instances, modules, etc. A *Contract* (i.e., a cyber contract) in our approach differs from the classic view of contracts proposed by Meyer in his DbC paradigm: DbC contracting parties can only be classes and methods whereas ours are not limited by their nature (instances, set of classes, programming units, etc.).. Furthermore, the *-* cardinality in the relation between the *Contract* and *ContractingParty* concepts enables the definition of a *Contract* involving multiple *ContractingParties*, potentially of different nature. This relation also provides

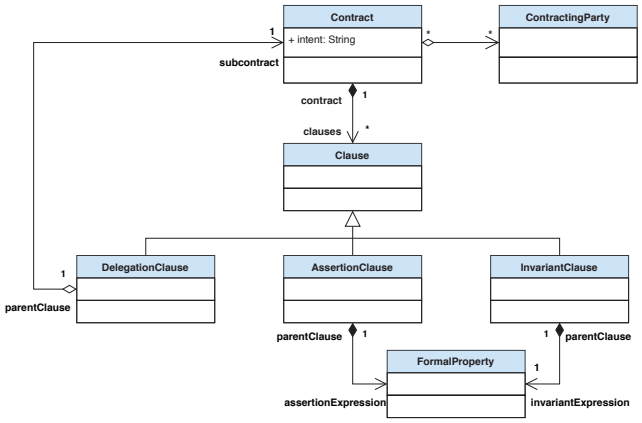

Figure 2: Cyber contract metamodel

the possibility to define multiple contracts for a given *ContractingParty*. In our cyber contract context, the *ContractingParties* stand for the entities involved in the security pattern definition.

Every *Contract* is defined as a set of clauses that structure the *FormalProperties* of the corresponding contract *Clauses*. There are three types of *Clauses* that can be added to a *Contract*: *AssertionClause*, *InvariantClause* and *DelegateClause*. A *FormalProperty* ensured by the contract refers to one of the two first type of clauses as showed the figure 3 on the Authenticator example. The *AssertionClauses* enable the specification of assertions that need to be verified at a certain time. They extend the notions of precondition and postcondition of the DbC approach in a sens that they can be expressed using temporal logic formulas. The *InvariantClauses* are assertions that must always be true, and can therefore be expressed with classical logic. The *DelegateClause* is a concept allowing for responsibilities of a contract to be divided into one or several subcontracts. This mechanism is typically used in the object-oriented paradigm to decompose a contract on the pattern classes, or any decomposition unit of the targeted language.

The last class of the CyberContract metamodel represents the concept of *FormalProperty*. It is an abstraction representing any kind of logic property that one could want to enforce using a contract. In a goal to remain independent of the property expression language, the metamodel does not require any precise expression language.

3.2 Security pattern cybercontract

Our approach allows for the definition of design pattern contracts. A design pattern is a set of classes and the description of their interactions. We argue that this description can be formalized using a design pattern contract. Such a formalization is particularly relevant regarding security design patterns as presented in Section 2.1.2.

We have identified two types of formal properties that are relevant to a pattern contract definition: functional and implicit properties.

Functional properties are all the properties expressing the expected behavior of a given pattern. They are the formalization of the interaction of classes as described in the pattern definition. When a pattern is described in natural language, some of the words that

are used come with an implicit meaning that can be crucial to the pattern. All these implicit elements, that we call implicit properties, need to be formally expressed in the pattern contract for it to be relevant. A basic example of implicit property is the notion of identifier. Most security design pattern describe fields as identifiers for certain classes. Such identifiers are obviously expected to identify instances of the class. We can for instance define a formal property stating that different instances must have different identifiers. This is an implicit but crucial property.

Thus, as presented in the metamodel, a cyber contract is defined as a set of functional and implicit security properties. The expression of these properties are based on the elements defined in the pattern including classes, instances, attributes and methods.

3.3 The Authenticator pattern contract

Let's exemplify the definition of a cybercontract for the Authenticator pattern. To do so, we will use *FormalProperty* expressions based on the object-oriented paradigm for the contract properties.

The *Authenticator* pattern describes a generic authentication mechanism. Six properties have been identified to ensure the security of such a pattern. The first four are implicit properties and the last two are functional properties.

(1) The first property to check is the uniqueness of the (authentication information, subject) pairs. This property is implicit in the pattern definition but is fundamental. Indeed, if it is not verified, the pattern has no more meaning since an *authentication information* would not identify a unique subject. With $I_{Subject}$ containing all instances of the *Subject* class, the property can be expressed as follows.

$$P1 : \forall a, b \in I_{Subject}, a \neq b \implies a.authInfo \neq b.authInfo$$

(2) Similarly, since the authentication information is linked to the subject instance, it must not vary during execution (probable identity theft). Let $a.authInfo_{ini}$ be the initial value of the field $authInfo$ of the $Subject$ instance a. Formally:

$$P2 : \forall a \in I_{Subject}, a.authInfo = a.authInfo_{ini}$$

(3) A major flaw that can jeopardize authentication systems concerns the non-integrity of the authentication authority. In order to be authenticated, a subject must make a request to the Authenticator. If an attacker succeeds in forging its Authenticator, he becomes master of the authentication system. It is therefore essential that the Authenticator cannot be modified. Thus, let $a.authenticator_{ini}$ be the initial value of the field $authenticator$ of the $Subject$ instance a. This property can then be expressed as follows.

$$P3 : \forall a \in I_{Subject}, a.authenticator = a.authenticator_{ini}$$

(4) At all time, the proof of identity of every subject must be either valid or undefined.

$$P4 : \forall a \in I_{Subject}, (a.idProof = \emptyset) \vee (a.idProof = a.authenticator.request(a.authInfo))$$

(5) After authentication, the proof of identity must be the one returned by the request method with the correct parameters. According to the DbC approach, it means that the following property should be ensured by the *authenticate* method of the *Subject* class.

$$P5 : self.idProof = self.authenticator.request(self.authInfo)$$

(6) The proof of identity returned by the *request* method of the *Authenticator* class must be valid if and only if the subject requesting it is who it claims to be. More formally, it means that the *request* method must ensure the following property. Let $\emptyset$ be the default value for a variable. The postcondition can then be expressed as follows.

$$P6 : self.check(authInfo) \vee returnValue = \emptyset$$

```
<Contract>
    <binding "Authenticator_pattern" />
    <clauses>
    <InvariantClause P1/>
    <Subcontract "SubjectContract">
        <binding "Subject" />
        <InvariantClause P2 ∧ P3 ∧ P4/>
        <Subcontract "authenticateContract"s>
            <binding "void authenticate()">
            <ensures P5/>
        </Subcontract>
    </Subcontract>
    <Subcontract "AuthenticatorContract">
        <binding "Authenticator"/>
        <Subcontract "requestContract">
            <binding = "ProofOfIdentity request(
     AuthenticationInformation authInfo)">
            <ensures P6/>
        </Subcontract>
    </Subcontract>
    </clauses>
</Contract>
```

$$P1 : \forall a, b \in I_{Subject}, a \neq b \implies a.authInfo \neq b.authInfo$$
$$P2 : \forall a \in I_{Subject}, a.authInfo = a.authInfo_{ini}$$
$$P3 : \forall a \in I_{Subject}, a.authenticator = a.authenticator_{ini}$$
$$P4 : \forall a \in I_{Subject}, a.idProof = \emptyset \vee$$
$$a.idProof = authenticator.request(authInfo)$$
$$P5 : self.idProof = self.authenticator.request(self.authInfo)$$
$$P6 : self.check(authInfo) \vee returnValue = \emptyset$$

Figure 3: XML representation of the Authenticator pattern contract

Figure 3 shows a simplified XML representation of this contract. This representation illustrates an instantiation of our metamodel on the Authenticator cyber contract. The binding with the Authenticator pattern is explicit and we can see the property P1 as an *InvariantClause* defined at the pattern scope. The delegation mechanism provided with the *DelegateClauses* is also illustrated within this example. Some of the previous properties are indeed within the scope of a single method (P5 and P6). They are thus delegated to the relevant classes using a *DelegateClause* (keyword *Subcontract*) and then to the correct methods.

4 EXPERIMENTS AND PRELIMINARY EVALUATION

In this section, we show how our approach can be mapped on two frameworks to specify cyber-contracts on security patterns and we show how to use them on existing code.

In particular, our approach was implemented in two software specification languages. Both implementations enable the addition of an extra layer of trust in the existing code, to enforce the formal properties of predefined security design patterns, without the need to modify the code. We implemented our approach in the Java Modelling Language (JML) [11], a DbC-based specification language, and PAMELA, a Java modelling language and framework [19].

These two languages were selected on the basis of an existing active community, the relevance regarding our approach and their ability to be extended without having to enrich or modify the syntax of the hosting language.

4.1 Java modelling language experiment

Java Modelling Language (JML) [5, 11] is a specification language that enables engineers to write DbC assertions as comments in Java code. These comments are then parsed by a specific compiler (called JMLC). The resulting java bytecode is the aggregation of java compile code and java assertions enforcing JML expressions. The JML language allows referencing the namespace of the Java program and the logical operators of the Java language. It also has some keywords and symbols that correspond, for the most part, to the concepts of the DbC approach. For example:

- *forall*, *exist*, => and <=>, which correspond to the universal quantification, existential quantification, and logical implication and equivalence, respectively;
- *invariant*, *requires* and *ensures* are used to represent, respectively, the invariants, preconditions and postconditions of contracts;
- assignable <*name*> to specify that a variable can be assigned in the method it specifies;;
- *old*<*name*> to reference the value of a variable before the call of the method it specifies;
- *result* to reference the return value of the method it specifies;
- *signals* describes the exceptions thrown by the method it specifies;
- *pure* specifies that the specified method does not have side effects;
- *also*: declares that a method inherits JML specification (preconditions and postconditions) from its supertype and adds specific specifications.

We use OpenJML [5] as the tool support to create the logical annotations in existing Java programs. In addition, it enables static or run-time checking of the validity of the annotations through static code verification and dynamic assertion checking capabilities.

The rest of this subsection shows how to specify the *Authenticator* pattern as a cyber-security contract with JML and some lessons learned from this experience.

To specify the *Authenticator* pattern as a cyber-security contract with JML, the six formal properties presented in Section 3.3 are implemented through JML expressions, as presented below:

(1) Uniqueness of authentication information

$$P1 : \forall a, b \in I_{Subject}, a \neq b \implies a.authInfo \neq b.authInfo$$

This property illustrates one of the limitations of the standard DbC approach and consequently the JML framework, in which a property is limited to be expressed for the scope of a class or a method, and so relative to one instance. As a result, this property cannot be expressed in JML since a JML expression is limited to the scope of a class and cannot involve more than one instance of the corresponding class.

(2) Invariance of authentication information

$$P2 : \forall a \in I_{Subject}, a.authInfo = a.authInfo_{ini}$$

This property is conceptually an invariant of the *Subject* class. It however requires to know, at all time, what is the initial value of the *authenticationInformation* field. One way to implement this "memory" in JML is to add the following ensure clause to every method, except the constructor, of the *Subject* class. The constructor call does not support the use of \old(authInfo) keyword. This is why we cannot use an invariant clause.

@ ensures authInfo == \old(authInfo);

(3) Invariance of authenticator

$$P3 : \forall a \in I_{Subject}, a.authenticator = a.authenticator_{ini}$$

This property can be implemented similarly to the previous one with the following ensure clause for every method of the *Subject* class.

@ ensures authenticator == \old(authenticator);

(4) Validity of the Proof of Identity

$$P4 : \forall a \in I_{Subject}, (a.idProof = \emptyset) \vee (a.idProof = a.authenticator.request(a.authInfo))$$

This property can be transposed to the following JML invariant for the *Subject* class.

@ invariant idProof == null || idProof == authenticator.request(authInfo);

(5) Correctness of authenticate method

$$P5 : self.idProof = self.authenticator.request(self.authInfo)$$

This property is directly a JML postcondition for the *authenticate* method of the *Subject* class.

@ ensures idProof == authenticator.request(authInfo);

(6) Correctness of the request method

$$P6 : self.check(authInfo) \vee returnValue = \emptyset$$

This property is also a JML postcondition for the *request* method of the *Authenticator* class.

@ ensures check(authInfo) || \result == null;

Regarding our approach, the previous example highlights the limitations of the standard DbC approach and consequently the JML framework. More specifically, there are two main expression issues that cannot be solved. The first is the limitation of a JML contract to the scope of a class or a method . This restriction prevents the expression of any contract property involving more than one instance of a class. The second issue has to do with the lack of memory of

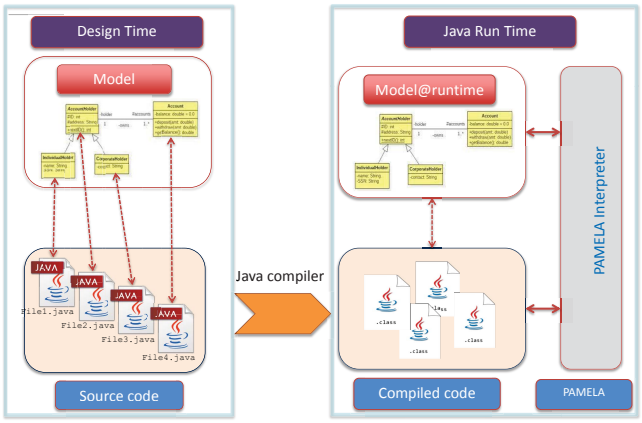

Figure 4: Overview of the PAMELA approach

the JML-based DbC approach. Indeed, in JML, a DbC property can only refer to the current state of an instance, or in the case of the post or precondition of a method, to the state before the call of the method. The previous examples show that we sometimes need to have an evolving state for the scope of the contract properties. This could be managed for instance by implementing pattern objects that could enforce the contract properties.

4.2 The PAMELA experiment

PAMELA is a Java modelling framework developed by the Openflexo community [1]. This project aims at bridging the gap between software modelling and code implementation.

In this context, PAMELA modelling framework proposes an approach where models are directly weaved in source code, by means of Java annotations. Thus, it does not require code generation. That way, both model and source-code coexist in the same artifact.

Figure 4 illustrates the PAMELA framework where we have at design time (left side of the figure), the model weaved in source code files based on the Java annotations and at run-time (right side of the figure) the PAMELA interpreter ensuring the link between the model behavior and the application Java byte-code.

Resulting execution is a combination of (i) executing plain Java byte-code as the result of the basic compilation of source code, and (ii) an embedded PAMELA interpreter executing PAMELA model semantics.

From an implementation standpoint, PAMELA uses *javassist* reflection library including the `MethodHandler` mechanism. Based on this support, the Java dynamic binding is overridden to provide the call of the PAMELA model behavior when an object method is invoked.

PAMELA framework offers some Aspect-Oriented Programming (AOP) features enabling the definition of additional behaviour to existing Java code.

We chose to extend the PAMELA framework to include our notion of *Pattern*, i.e. a composition of multiple classes, known as *Stakeholders*, whose expected behavior is defined in a pattern contract, along with formal properties which must be ensured at run-time. More specifically, implementing *Patterns* with PAMELA provides:

- the ability to use existing plain Java code, without any restriction;
- the ability to monitor the execution of the code;
- the ability to offer extra structural and behavioral features, executed by the PAMELA interpreter;
- a representation of *Patterns* as stateful objects. Such objects can then evolve throughout run-time and compute assertions using any paradigms (e.g., LTL - Linear Temporal Logic);
- the ability of having multiple extension points. In other words, the ability to create new patterns, or to redefine or specialize existing ones.

Patterns are defined in PAMELA using three classes, each one representing a different conceptual level:

- a `PatternFactory`. This class is responsible for identifying, at run-time, the declared patterns in the Java byte-code.
- a `PatternDefinition`. This class represents an occurrence of the pattern in the supplied byte-code. It has the responsibility of maintaining links with all classes and methods involved in the pattern, as well as managing the life-cycle of its `PatternInstances`.
- a `PatternInstance`. This class represents the instance of a pattern at run-time. It is responsible for maintaining the pattern state and providing pattern behavior and contract enforcement.

To declare a *Pattern* on existing code, pattern elements such as *Pattern Stakeholders* and methods need to be annotated with provided pattern-specific annotations. These annotations will be discovered at run-time by the `PatternFactory` and stored in `PatternDefinition` attributes.

To validate the implementation of our approach in PAMELA we propose an implementation of the *Authenticator pattern* as a cybersecurity contract. Figure 5 presents the previous class structure in the case of the Authenticator pattern. Note that each attribute of the `AuthenticatorPatternDefinition` class has a corresponding annotation (displayed as a comment).

Figure 6 presents the composition of the *Authenticator* pattern with an existing base of code. The *Authenticator pattern* requires the definition of two *stakeholders* (*Authenticator* role and *Subject* role) which have to be played by instances of provided Java classes. A set of annotations coming with the *security pattern* definition is used to explicit that roles (respectively `@Authenticator` and `@AuthenticatorSubject` annotations). The definition of the pattern also requires the distribution of responsibilities and relationships according to the underlying semantics of the pattern (here, the authentication concerns). The request authentication method is identified using the `@RequestAuthentication` annotation. In the same way, we must identify some responsibilities in the `Client` class: the method providing the *Authenticator* access, the method providing the authentication information, the method setting the proof of identity and the `authenticate()` method itself.

The excerpt of code presented Figure 7 shows how the Authenticator pattern is used by means of annotations to an existing base

[1] https://www.openflexo.org

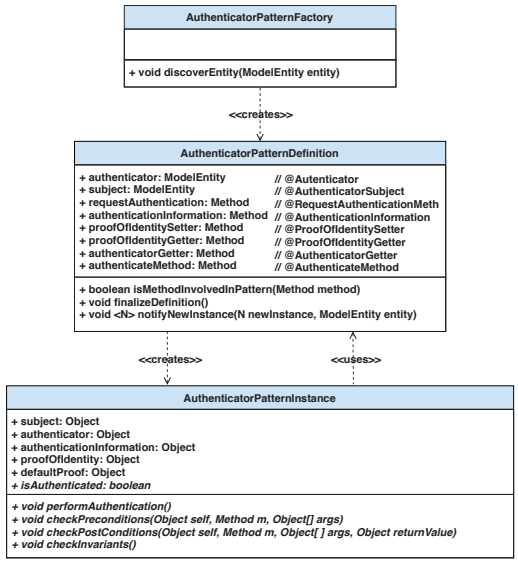

Figure 5: PAMELA Authenticator pattern class diagram. Emphasized attributes and methods are the one referenced in Figure 8.

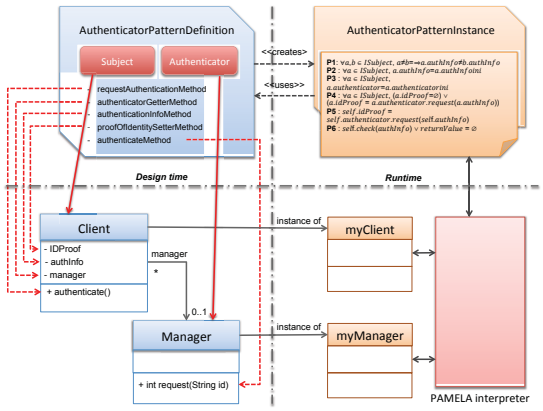

Figure 6: PAMELA vision of the authenticator pattern

of code. An instance of the `Manager` class plays the *Authenticator* role, while an instance of the `Client` class plays the *Subject* role.

At run-time, both functional code and security pattern logic, i.e pattern behavior and contract enforcement, are weaved by the PAMELA framework.

This implementation, unlike JML, provides a contract enforcement mechanism which is hard-coded within PAMELA *Pattern* classes. This abstraction is closer to the requirements of the end-user, who may want to add an authentication mechanism to an existing code. The user only need to annotate his code and PAMELA will automatically handle the pattern business logic (both, pattern behavior and assertion checking).

```
@ModelEntity
@Authenticator(patternID = "MyPatternId")
public class Manager {

  @RequestAuthentication(patternID = "MyPatternId")
  public int request(@AuthenticationInformation(patternID
      = "MyPatternId", paramID = "id") String id) {
    return ...;
  }
}

@ModelEntity
@AuthenticatorSubject(patternID = "MyPatternId")
public class Client {

  public Client(Manager authenticator, String id) {
    ...
  }

  @AuthenticationInformation(patternID = "MyPatternId",
     paramID = "id")
  public String getAuthInfo() {
    return ...;
  }

  @ProofOfIdentityGetter(patternID = "MyPatternId")
  public int getIDProof() {
    return ...;
  }

  @AuthenticatorGetter(patternID = "MyPatternId")
  public Manager getManager() {
    return ...;
  }

  @AuthenticateMethod(patternID = "MyPatternId")
  public void authenticate() {
    setIDProof(getManager().request(getAuthInfo()));
  }

  @RequiresAuthentication
  public void securedMethod() {
    ...
  }
}
```

Figure 7: Example showing how to use the Authenticator pattern defined as a PAMELA cyber-contract

Figure 8 depicts the control flow of the execution of a method annotated with `@RequiresAuthentication`. This annotation is used to identify methods which must trigger the authentication process before being executed. The call will thus be handled as follows:

- The method is identified as a method of interest because this method requires authentication.
- The method is passed to the relevant `AuthenticatorPatternInstance` (second object on the Figure) before the method execution.
- The authentication process is to be executed, because subject is not authenticated yet (`isAuthenticated` is `false`).
- Before executing the called method, contract invariants are checked (properties P1 to P4) via the `checkInvariants()` call. If a clause breach is detected, an exception is thrown

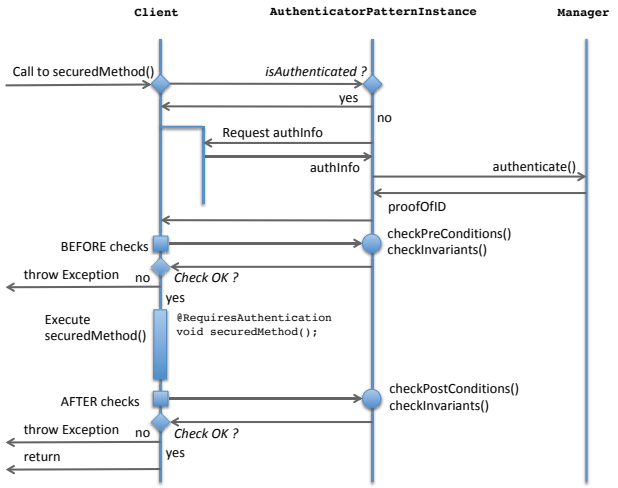

Figure 8: Authenticator control flow

and the method is not executed. This is also at this time that preconditions are checked (method `checkPreConditions`).
- The method is invoked.
- The method is once again passed to the `AuthenticatorPatternInstance` after the method execution. Contract invariants are checked. If a clause breach is detected, an exception is thrown. Similarly, post-conditions are checked (method `checkPostConditions`).
- Finally, the result of the method is returned to the caller.

4.3 Lessons learned from experiments

The two experiments emphasize that the formal behavioral definition of security contracts is an efficient approach to provide several implementations based on our design by contract definition.

The JML experiment highlights the limitations of this DbC implementation for security pattern contract definition. DbC implementations are too limited and fail to express properties requiring multiple class instances or an evolving pattern state. This justifies the necessity to extend the scope of contract definitions to more complex contracting parties, such as sets of classes.

These limitations are overcome with the PAMELA approach which provides the ability to freely build models based on multiple classes with their own semantics and behavioral code. In this case, our security cyber-contracts are applied at the top of the set of classes used in the security pattern.

Regarding the challenges we wanted to target, PAMELA provides an efficient and easy framework to apply security patterns using Java annotations . This can be done at a very early stage of design process of a software component, but this framework can also be applied on legacy source code beeing annotated.

Securing existing code remains a difficult task but our proposal, based on both formal behavioral contract and run-time property enforcement, enforces the security of software components.

5 RELATED WORK

The security by design approach is mainly based on security patterns applied to many phases of the development process. Several publications focus on security patterns, in general or specific domains such as Yoder and al. [21], Fernandez-Buglioni [7], and Yoshioka et al. [22]. In most cases, the patterns are defined based on the original approach of the Gang of Four [8] with textual definitions and usually represented with refined and stereotyped UML diagrams (Class diagrams for the structural part and Sequence diagrams to represent the behavior of the patterns).

Formal definitions of security patterns are also provided by several authors [4], [2], [6]. In most cases, the purpose of these formal definitions is to analyze the behavior of the patterns at design level or enabling automatic analysis operations. Our proposal to formalize security patterns with a design by contract approach aims at ensuring a secure implementation of patterns. Therefore, this work complements the previous efforts to formalize security patterns.

Originally, the design by contract approach was defined to support reliability on object oriented applications [14]. Then, it was also extended to component modeling [3] and formal definitions [15]. At code level, the DbC approach is implemented with the JContractor [10] framework, which provides run-time contract checking by instrumenting the bytecode of Java classes that define contracts. It however remains limited to the initial scope of Meyer's contracts; i.e., classes and methods. The closest approach to our work is the Aspect Oriented Programming (AOP) approach used by Hallstrom and al. [9] to monitor the pattern contract. Each pattern contract is associated with a dedicated pattern and is defined in an *aspect*. This *aspect* is used to monitor, at run-time, the applied pattern. Nevertheless, this approach is exclusively focused on the implementation without a will to provide abstraction related to the contract definition. Thus, our formal definition could be used to target such AOP implementations by providing the missing abstraction.

To the best of our knowledge, the design by contract approach extended to security patterns, associated with a formal definition of the security properties was never used to improve the security of software system, at source code level.

6 CONCLUSION

In this paper, we presented a design by contract approach to formalize security patterns. This novel approach improves the behavioral definition of the security patterns and therefore enforces the security of systems at the source code level.

The ambition of this article was to address two issues regarding the security of code through the use of security patterns. Our formalization of both functional and implicit parts of security design patterns allowed us to address the first challenge. Our contracts thus provides an explicit description of security patterns which captures tacit elements of the pattern, usually undocumented. The second targeted issue is largely addressed through our PAMELA experiment. We indeed provided an implementation allowing for run-time contract enforcement based on code annotation. Such active monitoring contributes to ensuring pattern security properties in legacy code.

As a future work, we plan to investigate about the use, strengths and weaknesses of our approach with real cases (e.g., Web applications exposed to cyber attacks) to evaluate its scalability and effectiveness.

REFERENCES

[1] Germán H Alférez, Vicente Pelechano, Raúl Mazo, Camille Salinesi, and Daniel Diaz. 2014. Dynamic adaptation of service compositions with variability models. *Journal of Systems and Software* 91 (2014), 24–47.

[2] Anika Behrens. 2018. What are Security Patterns? A Formal Model for Security and Design of Software. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*. 1–6.

[3] Antoine Beugnard, J-M Jézéquel, Noël Plouzeau, and Damien Watkins. 1999. Making components contract aware. *Computer* 32, 7 (1999), 38–45.

[4] Betty HC Cheng, Bradley Doherty, Nick Polanco, and Matthew Pasco. 2019. Security Patterns for Automotive Systems. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 54–63.

[5] David R Cok. 2014. OpenJML: software verification for Java 7 using JML, OpenJDK, and Eclipse. *arXiv preprint arXiv:1404.6608* (2014).

[6] Luis Sérgio da Silva Júnior, Yaan-Gael Guéhéneuc, and John Mullins. 2013. *An approach to formalise security patterns.* Technical Report. Citeseer.

[7] Eduardo Fernandez-Buglioni. 2013. *Security patterns in practice: designing secure architectures using software patterns.* John Wiley & Sons.

[8] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software.* Pearson Education India.

[9] Jason O Hallstrom, Neelam Soundarajan, and Benjamin Tyler. 2004. Monitoring design pattern contracts. In *Proc. of the FSE-12 Workshop on Specification and Verification of Component-Based Systems*. 87–94.

[10] Murat Karaorman and Parker Abercrombie. 2005. jcontractor: Introducing design-by-contract to java using reflective bytecode instrumentation. *Formal Methods in System Design* 27, 3 (2005), 275–312.

[11] Gary T Leavens, Albert L Baker, and Clyde Ruby. 2006. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes* 31, 3 (2006), 1–38.

[12] Raúl Mazo and Christophe Feltus. 2016. Framework for Engineering Complex Security Requirements Patterns. In *2016 6th International Conference on IT Convergence and Security (ICITCS)*. IEEE, 1–5.

[13] Raúl Mazo, Carlos Jaramillo, Paola Vallejo, and Jhon Medina. 2020. Towards a new template for the specification of requirements in semi-structured natural language. *Journal of Software Engineering Research and Development* 8 (2020), 3–1.

[14] Bertrand Meyer. 1992. Applying'design by contract'. *Computer* 25, 10 (1992), 40–51.

[15] Sebti Mouelhi, Mohamed-Emine Laarouchi, Daniela Cancila, and Hakima Chaouchi. 2019. Predictive Formal Analysis of Resilience in Cyber-Physical Systems. *IEEE Access* 7 (2019), 33741–33758.

[16] Amina Souag, Raúl Mazo, Camille Salinesi, and Isabelle Comyn-Wattiau. 2016. Reusable knowledge in security requirements engineering: a systematic mapping study. *Requirements Engineering* 21, 2 (2016), 251–283.

[17] Amina Souag, Raúl Mazo, Camille Salinesi, and Isabelle Comyn-Wattiau. 2018. Using the AMAN-DA method to generate security requirements: a case study in the maritime domain. *Requirements Engineering* 23, 4 (2018), 557–580.

[18] Tithnara Sun, Bastien Drouot, Fahad Golra, Joël Champeau, Sylvain Guerin, Luka Le Roux, Raúl Mazo, Ciprian Teodorov, Lionel Aertryck, and Bernard Hostis. 2020. A Domain-specific Modeling Framework for Attack Surface Modeling. In *International Conference on Information Systems Security and Privacy*.

[19] Openflexo team. 2020. *PAMELA modelling framework.* https://pamela.openflexo.org

[20] Hironori Washizaki, Tian Xia, Natsumi Kamata, Yoshiaki Fukazawa, Hideyuki Kanuka, Dan Yamaoto, Masayuki Yoshino, Takao Okubo, Shinpei Ogata, Haruhiko Kaiya, et al. 2018. Taxonomy and Literature Survey of Security Pattern Research. In *2018 IEEE Conference on Application, Information and Network Security (AINS)*. IEEE, 87–92.

[21] Joseph Yoder and Jeffrey Barcalow. 1997. Architectural patterns for enabling application security. In *Proceedings of the 4th Conference on Patterns Language of Programming (PLoP'97)*, Vol. 2. Citeseer.

[22] Nobukazu Yoshioka, Hironori Washizaki, and Katsuhisa Maruyama. 2008. A survey on security patterns. *Progress in informatics* 5, 5 (2008), 35–47.