Reusable formal models for secure software architectures

Thomas Heyman, Riccardo Scandariato and Wouter Joosen
IBBT-DistriNet, KU Leuven
first.last@cs.kuleuven.be

***Abstract*—Formal modelling techniques are often disregarded as their semantics are too distant from the mainstream practice of software architecture design, which is dominated by the use of component based modelling and patterns. This paper advocates the need for formal modelling techniques for humans, i.e., software architects who need to precisely ascertain the security properties of their design models. We contribute a technique that enables architects to more easily construct verified, secure architecture designs by assembling already verified security pattern models. Our approach is illustrated with a pattern language for accountability. It is validated by an observational study that shows that the approach produces reusable results, and is able to uncover relevant architectural security flaws.**

***Keywords*-software architecture; security patterns; modelling**

I. Introduction

Software systems are essential for critical parts of our society. You might have read this (or a similar) sentence many times at the beginning of computer science papers. Often, the papers continue by suggesting a formal approach to the development of critical systems, as formal approaches are essential in building strong assurance that these systems uphold their intended properties such as security. In this respect, this paper is no exception. We present an approach to formally model software architectures intended for critical systems, e.g., electronic health record management systems and smart energy grids. Although they are not safety critical, these systems do require a higher level of assurance and justify the extra effort of adopting formal techniques. Especially in the early stages of the software development life cycle (including architectural design), formal modelling needs to be adopted as state of the practice, given the key role of these stages in realizing the required system qualities such as security—often a high priority in critical systems.

However, a second aspect is commonly swept under the rug. Usually, developers of more critical systems are of no different breed than the developers of less critical systems (e.g., the next social networking site). Still, the level of expertise required to master many of the present formal techniques goes beyond the skill set of the average software architect. This paper advocates the need for formal modelling techniques whose semantics are not too distant from the mainstream practice of software architecture design, which is dominated by the use of component based modelling and patterns. In particular security patterns are gaining popularity in the secure software engineering domain.

Our main contribution, presented in Section III, is a technique to rigorously model architectural security patterns in two connected formal models. The *refined model* contains the internals of the pattern and is defined by a security expert who undergoes the task of interpreting the pattern documentation, translating it into a sound formal model, and verifying its security properties. The complexity of the refined model is not exposed to the architect. Rather, the refined model is simplified to an equivalent *abstract model*, which is easier to both understand and use. This model is directly used by the architect, who can more easily integrate it in larger models (the whole design) to reason about compositional security properties. We achieve usability by 1) keeping models close to architectural concepts and integrating well with architectural design processes, as we also support reasoning about partial models common in the early stages of architecture design, and 2) by packaging patterns into understandable, reusable models. The technique is illustrated by modelling a pattern language, i.e., a self-contained set of five architectural security patterns, for the domain of accountability. The usability of the formal models is validated in Section IV through an observational study involving two subjects who each solved two design tasks using abstract pattern models. The subjects successfully completed the tasks in a reasonable amount of time, and reported high satisfaction levels for usability.

The second contribution, presented in Section V, is geared towards the security patterns community. While creating the refined models, the authors gained insight into these five patterns and discovered gaps in their documentation. Obviously, insufficient documentation forces the architects to interpret the pattern by themselves, thereby opening up the possibility for design flaws and impairing the assurance that the patterns provide. The uncovered gaps are discussed in this paper with the aim at improving the quality of the pattern documentation. Our technique could be applied systematically to the body of security patterns in order to unearth even more issues.

Our results are discussed in Section VI and compared to related work in Section VII. We conclude in Section VIII. First, Section II introduces previous work on which our modelling technique is based.

II. Background on formal modelling

Section II-A summarizes our approach to model and verify the security of software architectures, which is documented in previous work [1]. Section II-B overviews how these models are abstracted.

A. Component-based modelling

We leverage Alloy [2], a first order logic based language that supports both visualising models and verifying their properties up to a specific scope. One advantage of model finders such as Alloy over traditional model checking tools is that they allow reasoning about partial models. The architect is able to specify only a subset of constraints that the resulting system-to-be should uphold, while the model finder will then reason about every possible instantiation of the system that is allowed by these constraints. Consider the following Alloy code.

```
1  // Abstract Logger
2  sig Message {}
3  one sig Log, Read in Operation {}
4  sig Logger in Component {
5    contents: Message→Time
6  }{
7    all c:Component,m:Message,r:Int,t:Time {
8      Execute[c,this,Log,m,t] ⇒ this.log[m,t]
9      Execute[c,this,Read,m+r,t] ⇒ this.read[m,r,t]
10 } }
11 pred Logger.log(m:Message,t:Time) { m in this.contents.t }
12 pred Logger.read(m:Message,result:Int,t:Time) {
13   m in this.contents.t implies result = 1 else result = 0
14 }
15 sig LoggerClient in Component {
16   currentLog: lone Logger →Time
17 }
```

This (part of an) architectural description introduces five domain specific types: one atomic type Message (line 2), two operation types Log and Read (line 3), and two component types Logger (line 4) and LoggerClient (line 15). The keyword **in**, similar to subtyping, indicates that every Logger and LoggerClient are also in the set of Components and will be subject to the same constraints as the Component super type.

The Component type is part of the meta model we have defined in previous work [1], which supports component based modelling. To facilitate modelling architectures, all generic architectural concepts such as Component, Connector, or Operation have been encoded in a reusable domain independent meta model. The meta model also defines the semantics of operation invocation and execution—an invocation only leads to execution when a corresponding Invocation is transmitted through a Connector that connects the caller to the receiving component. Additionally, the meta model links Components to a Node on which they are deployed. Nodes, in turn, are connected by Links. The language defined in the meta model therefore suffices to model logical architectural views (the component view), scenario views (how certain use cases are executed by the system), and physical views (how the software is mapped on hardware).

In the example, a relationship called currentLog is introduced on the type LoggerClient (line 16, similar to a class attribute, which relates a LoggerClient to zero or one Loggers. Similarly, the relation contents (line 5) relates every Logger to a time dependent set of Messages. Note that time is modeled explicitly in Alloy, so the suffix →Time makes that relation dependent on Time and, hence, mutable (time independent relations maintain the same value, and are therefore immutable). This partial model allows reasoning about architectures consisting of LoggerClients that each have a reference to a Logger. It does not specify how many LoggerClients or Loggers there should be, or whether every LoggerClient has its own Logger. The analysis will take all the situations not excluded by the specification into account.

The behavior of a component is defined by the semantics of its operations. Every component offers an interface which consists of one or more Operations (e.g., Log and Read on line 3), which other components can invoke. For every operation, a predicate (e.g., log on line 11 and read on line 12) is defined on that component that models the postconditions of that operation. These predicates become true whenever that operation is executed (line 8). This way of modelling corresponds to message passing, but our models also support interception (i.e., triggering operations based on observing an invocation that was not necessarily destined for that component), as well as multicast (i.e., multiple receivers of the same invocation).

The goal of the analysis is to verify whether the architectural model upholds certain security requirements. Security requirements can be encoded as predicates over the operations offered by a component (i.e., its interface). This decouples the requirement from the implementation details of the component, and will allow us to modify its internals without needing to update the requirement. For instance, the security requirement of the Logger, 'once logged, every message can forever be read back', is encoded as follows.

```
18 // Security requirement
19 assert LoggingRequirement {
20   all l:Logger,m:Message,t:Time | l.log[m,t] implies
21     all future:t.nexts | l.read[m,1,future] }
```

In the initial model as presented, the LoggingRequirement does not hold yet. In that case, Alloy will provide the architect with a counterexample, i.e., an instantiation of the architectural model that upholds all modelled constraints, yet violates the requirement. The architect can then mitigate that counterexample in one of three ways. First, the counterexample might show that the model does not functionally behave as the architect had in mind. This indicates a flaw in the architecture, and should be resolved by changing the description of the relevant model elements (e.g., changing an incorrectly specified operation).

Second, the architect might find nothing wrong with the counterexample, hinting that the security requirement is underspecified and classifies benign situations as insecure. In that case, the counterexample should be used to revise the security requirement.

Third, the counterexample might be functionally correct, but hint at a weakness that the architect did not foresee. For instance, currently the Logger does not enforce that Messages in the contents relation for a specific Time, remain in the contents relation for the next Time. In other words, messages stored by the Logger could be trivially deleted. This indicates a missing trust assumption (in this case, on the underlying storage medium on which the Messages are stored by the Logger), and should be mitigated by adding an *explicit assumption* to the model. For instance, the architect might explicitly add the constraint that Messages are never removed from the contents relation, implying that the storage medium is trustworthy.

```
22 // Explicit assumption
23 fact StorageIsTrustworthy {
24   all l:Logger,m:Message,t:Time |
25     m in l.contents.t ⇒ m in l.contents.(t.next)}}
```

These externalized constraints are precisely the trust assumptions on which the security of the modelled architecture depends.

B. Abstracting and composing a model

By using refinement, a detailed part of an architecture can be replaced with an equivalent abstraction. The full details of our refinement process are out of scope for this work, however, we use these abstract models to reason about compositional properties and refined models to verify local security requirements and elicit trust assumptions down to an arbitrary level of detail.

Consider the Logger introduced in Section II-A. While it behaves as an ideal Logger of which no message can be deleted, it might be difficult for the architect to accept the corresponding trust assumption StorageIsTrustworthy on the underlying storage medium. In order to remedy this, the architect can refine the Logger to assign sequence numbers to every logged Message.

```
26 // Refined Logger
27 sig NumberedMessage { id: one Int, contents: one Message }
28 sig Logger in Component {
29   nextUID: Int one →Time,
30   contents: NumberedMessage →Time
31 }{...
32   all c:Component,r:Int,t:Time {
33     Execute[c,this,Verify,r,t] ⇒ this.verify[r,t]
34 } }
35 pred Logger.log(m:Message,t:Time) {
36   some nm:NumberedMessage {
37     nm.contents = m and nm in this.contents.t
38     nm.id < calculateNextUID[this,t]
39 } }
40 pred Logger.verify(result:Int,t:Time) {
41   this.allEntriesAccountedFor[t] and this.noExtraneousEntries[t]
42     implies result = 1 else result = 0
43 }
```

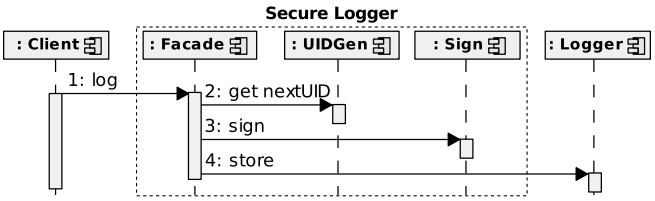

Figure 1. Sequence diagram of secure logging (adapted from [3]).

The refined Logger introduces a counter, nextUID, and a third operation, Verify, to check whether the Logger contains a Message for every sequence number up to the counted amount of logged messages (and no more).

We can now verify whether every operation of the abstract Logger is equivalent to the refined one, e.g. for the log operation, l.R/log[msg,t] ⇔ l.A/log[msg,t] (where R/log denotes the refined log operation, and A/log the abstract). The equivalence holds, given the following three assumptions. First, the nextUID attribute should always be the exact number of messages already logged. Second, both NumberedMessage ids and contents should be unique. Third, the Logger should always verify (i.e., the result of the verify operation should always be '1'). As the security requirements are expressed over the shared operations (which behave equivalently), the refined Logger is secure whenever the abstract one is. In effect, the architect has operationalized the original trust assumption on the storage medium (line 23) to checking the result of the Verify operation. The architect can now use this refined model to assess whether the resulting three assumptions are acceptable in the context of the system being modelled, or if the model should be additionally refined. The abstract model can then be used to verify properties on the composition of the Logger and the rest of the system, while uncovering trust assumptions on the compositional level.

III. Modelling architectural patterns

To show how this approach can be applied to model security patterns, we use the Secure Logger architectural security pattern [3]. It improves upon the Logger from Section II-A by adding digital signatures to the numbered log entries to remove the assumption that message ids and contents are unique (which is another way of expressing that no one else could 'fake' a Message with a specific id). Its behavior is shown in Figure 1.

Following the guidelines from the pattern description, the Secure Logger can be modelled as follows. The SecureLogger component stores a list of SignedMessages, which are abstractions of digitally signed NumberedMessages from Section II-B. The rest of the SecureLogger is similar to the

regular Logger, except that the Log operation wraps the new NumberedMessage in a SignedMessage at the time of logging, and that the Verify operation also checks the integrity of every SignedMessage.

```
44 sig SecureLogger in Component {
45   contains: SignedMessage→Time, nextUID: Int one→Time
46 }
47 pred SecureLogger.verify(result:Int,t:Time) {
48   this.allEntriesAccountedFor[t] and this.noExtraneousEntries[t]
49     and this.entriesHaveValidSignature[t] implies
50       result = 1 else result = 0
51 } ...
52 sig SignedMessage {
53   content: NumberedMessage one →Time,
54   signedContent: NumberedMessage one →Time,
55   signedBy: Component one →Time
56 }
57 pred SignedMessage.isValid(t:Time) {
58   this.content.t = this.signedContent.t
59   this.signedBy.t = SecureLogger
60 }
```

It turns out that every security pattern can be modelled similarly based on a handful of concepts.

A. Translating essential pattern concepts to Alloy

In general, security patterns can be modelled following the philosophy of [4], which decomposes these patterns in new components, roles, requirements, expectations, and residual goals, as depicted in Figure 2. This work contributes how these essential pattern concepts are translated to Alloy.

Newly introduced components are introduced solely to realize the pattern. The Secure Logger pattern introduces the SecureLogger as a new component. New components are modelled as new component types as per Section II. These components uphold *behavioral requirements*. The SecureLogger, for instance, should allow its users to log messages so that they cannot be altered or deleted, and so that events cannot be lost [3]. Behavioral requirements are encoded as verifiable assertions expressed over the interface of the new component(s), e.g., the LoggingRequirement from Section II-A.

Roles are named references that should be mapped to existing components. This allows the pattern writer to document how new components can be properly integrated in an existing architecture. The Secure Logger pattern mentions the Client role that sends requests to be logged, and the Logger role that stores the processed log entries in the backend. Roles can be modelled in two ways. First, they can be made explicit by introducing a new component type. For example, the LoggerClient on line 15 can be seen as the explicit introduction of a client role, on which additional constraints can be placed. Second, roles can be modelled implicitly—the client role for the Logger can be described as the set of all components that invoke the Log operation at some point.

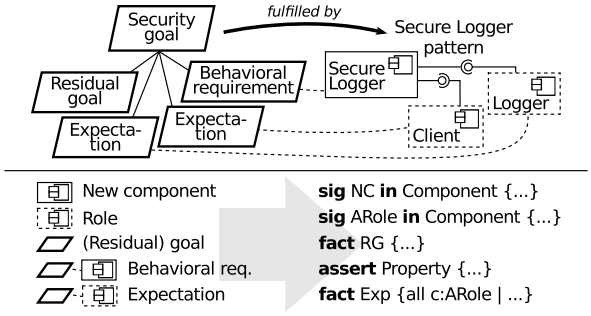

Figure 2. Modelling security patterns with Alloy (top part based on [4]).

```
61 fun ClientRoleSet : set Component { { c:Component |
62   some l:Logger,m:Message,t:Time | Invoke[c,l,Log,m,t] } }
```

Expectations are additional requirements imposed on the roles and their interaction with the new components. For instance, the Secure Logger pattern warns that care should be taken that messages are not altered while sent from the Client to the SecureLogger. Using our meta model, this would be expressed as the following fact.

```
63 fact ClientExpectation {
64   all l:Logger,c:ClientRoleSet,conn:getConnectors[c,l] |
65     conn.tamperProof }
```

Residual goals are other security considerations on which the correct operation of the new components depend, that fall outside the scope of the pattern. For instance, the Secure Logger pattern mentions that cryptographic keys should be properly managed. These residual goals should then be fed back into the requirements engineering process. Residual goals can be modelled as extra facts that are added to the model, and are assumed to hold.

Using these concepts to interpret and formalize the textual pattern documentation makes the translation less arbitrary and provides additional insights in trust assumptions.

B. Security patterns and trust assumptions

The trust assumptions that are elicited during the analysis process can be categorized as expectations and residual goals. Both constrain the environment of the pattern so it realizes its security requirement. When the assumption is predicated over a role or its interaction with the new components, it is an expectation. For instance, the Secure Logger assumes that the connectors connecting the Client to the Secure Logger are tamper proof. This is clearly a constraint on the interaction between the Client role and the SecureLogger. If the assumption is role independent, it is a residual goal. For instance, another assumption states that the signatures of the SecureLogger should be unforgeable. This is independent of any LoggerClient interaction.

Trust assumptions are present in the refined pattern models, as well as in the composition. The assumptions of the refined model document what the pattern requires of

its environment to realize its (local) security requirement. Clearly, this is crucial information for the users of those patterns (i.e., the architect) and the pattern community. The assumptions found by connecting the abstract model to the rest of the system, document what the composition requires to realize its security requirements. This feedback is essential for the architect to determine whether the composition (and, *a fortiori*, the architecture) is secure.

IV. Evaluation

We have modelled a language of five security patterns according to the technique of Section III. The trust assumptions found in the refined models are discussed in Section V. First, we report on the usability of using the abstract models to discover compositional assumptions, validated by an observational study in which two subjects had to extend an architecture with two patterns.

A. Participants

The two subjects participating in this case study were one researcher from the department of Computer Science of the KU Leuven and another from the university of Duisburg-Essen. Their expertise and relevant experience was established via an interview prior to starting the exercise. Both participants have seven years of experience in the field of software security, with one subject focussing on software architecture research and the second on requirements engineering. Furthermore, they have extensive experience with security patterns. Both participants have come into contact with formal modelling (one of the subjects with Alloy in particular), but it is not their main expertise. The participants were selected based on their similar experience, to remove potential bias in the results arising from different skills.

B. Experimental object

The case study is deliberately kept simple and as generic as possible. This way, the subjects do not need to become familiar with the problem domain. Additionally, it avoids bias through different levels of domain mastery. Finally, it allows the tasks to be performed within a three hour slot. The application to be extended by the subjects consists of a model of an online Shop module with an interface containing one operation PlaceOrder. This operation receives an Order, and adds it to the internal list of orders to be processed. Additionally, an external PasswordList contains combinations of Accounts and Password hashes. Only upon receiving valid login credentials is an Order processed. The model of the application contains one functional requirement that states that it should be possible for some customers to place orders.

```
66 pred FunctionalRequirement {
67   some s:Shop,c:Component,a:Account,p:Passwd,o:Order,t:Time |
68     s.placeOrder[c,a,p,o,t] }
```

This requirement is used later on to test that no contradictions were introduced during modelling.

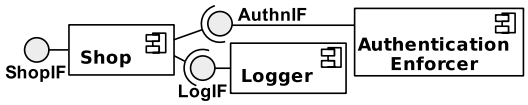

Figure 3. Structural view on the case study reference solution.

C. Tasks

The subjects were asked to complete three tasks in total. The *first task* consisted of a warm-up exercise, in which the subjects were given two security requirements and asked to make them hold in the Alloy model by adding trust assumptions, but without introducing any security patterns yet. The first security requirement stated that every time an order is placed, it can be read back at any later time.

```
69 assert SecurityRequirement1 {
70   all s:Shop,c:Component,a:Account,p:Password,o:Order,t:Time {
71     s.placeOrder[c,a,p,o,t] ⇒ all t1:t.nexts | s.hasOrder[o,t1]}}
```

The second security requirement stated that for every order that can be read back, a corresponding order was placed earlier by an authenticated caller.

```
72 assert SecurityRequirement2 {
73   all s:Shop,o:Order,t:Time | s.hasOrder[o,t] ⇒
74   some a:Account,p:Password,c:Component,t1:t.prevs+t |
75   s.placeOrder[c,a,p,o,t1] and a→p in PasswordList.contents }}
```

Additionally, the subjects were tasked with verifying that their solution still had functional instances. The purpose of the warm-up exercise was to familiarize the subjects with the initial model, while gaining some experience with the analysis process. The *second task* was to extend the model with the secure logger pattern to operationalize the first security requirement. The subjects were not allowed to change the security requirement or the signatures of the operations of the system. The *third task* was then to extend the resulting model with the authentication enforcer to realize the second security requirement. A UML representation of the Alloy model to be created is provided in Figure 3.

D. Setup

All three tasks were performed by both subjects in the same room during one supervised session. The subjects each worked on their own laptop computers, and were provided with an archive containing the following items: a JAR file of the Alloy Analyzer version 4.1.10, the architectural meta model as described in Section II-A, both the abstract and refined models of the secure logger and authentication enforcer patterns, the initial model of the online shop system (counting 147 lines of code), and a 'cheat sheet' with some domain independent code snippets on how to use the Alloy language. This material is available for reproducibility [5].

After the warm-up, the subjects were only allowed to ask specific modelling questions related to Alloy (e.g., "How do I model an if-else construction?", "What does this syntax error mean?"). Questions related to how the patterns should

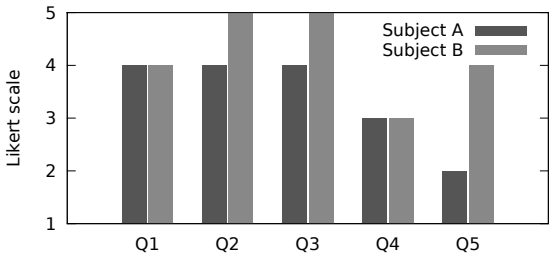

Figure 4. Answers to the questionnaire in Section IV-F.

be integrated, or open-ended questions (e.g., "What should I do here?") were not allowed. Additionally, the subjects were not allowed to collaborate.

E. Measurements

The subjects were timed during the task execution (not during the warm-up). After finishing all three tasks, the subjects were asked to send in their final models. A reference solution was created beforehand by the author most experienced in applying the approach. This was used as a baseline to evaluate the solutions of the subjects.

At the end, the subjects were asked to fill out a brief exit questionnaire asking about the perceived effort of applying the patterns, the usability of the abstract models, and the subjects' confidence in the security of the end result.

F. Observed results

Feasibility: Both candidates successfully extended the system without support from the authors (except for answering specific Alloy questions not related to integrating the patterns, as mentioned earlier). Subject A managed the tasks in one hour and three minutes, subject B in two hours thirty minutes. This can probably be attributed to subject A having more prior experience with Alloy.

Usability: Concerning the usability of the approach, the exit questionnaire contained the following questions. **Q1.** Did you find the library useless (score 1) or useful (score 5) while modelling? **Q2.** Would you prefer modelling everything from scratch (score 1) or use the library (score 5)? **Q3.** Did the library help convince you of the security of the system? (Score 1, no; score 5, yes.) **Q4.** From trivial (score 1) to impossible (score 5), how difficult was it to add a pattern? **Q5.** Did the pattern models make interpreting counterexamples easier (score 1) or harder (score 5)? The answers are shown in Figure 4. They indicate that both subjects perceived the pattern library as useful and increasing the confidence in the security of the system (Q1-Q3). The perceived effort of adding reusable pattern models is medium (Q4). There was no consensus on whether the pattern models helped in interpreting counterexamples (Q5).

Correctness: Both solutions were correct (i.e., they upheld the security requirements and correctly integrated the patterns), except for a small design flaw in solution A. Namely, after authentication the Shop does not verify whether the resulting Subject is valid. This pushes the responsibility to the environment and effectively makes verification of the Subject a residual goal. This was the only flaw found in either solution, and was present in the 'user' code integrating the pattern in the architecture (not in the pattern model).

Table IV-F is used as a checklist to compare the documented assumptions of both subjects to the reference solution. The assumptions are compared on the lowest shared level of granularity (e.g., the assumption E **and** F is seen as two individual assumptions E, F). A ✓ indicates that that solution contained the assumption. An exclamation mark (!) denotes that the assumption was present, but made superfluous by a later assumption. For instance, Assumption 5 was phrased by subject B as **all** s:Shop,t:Time | s.log.verify[1, t]. However, subject B then wrote Assumption 6 as Invoke[s,s.log,Log,o,t] **implies** s.log.read[o,1,t.next], which implies Assumption 5, making it obsolete. A question mark (?) denotes that that assumption was absent—it was not made explicit, but hard coded in the architectural model. For example, Assumption 2 was made implicit in solution B as the reference of the Shop to the SecureLogger was immutable[1]. Solution A contains all seven trust assumptions of the reference solution. It additionally contains Assumption 3, a stronger version of Assumption 4, making the latter superfluous. It expresses the residual goal introduced by the authentication flaw discussed earlier. Five assumptions of Subject B are also present in the reference solution. Solution B contains one extra assumption, Assumption 9, due to a slightly weaker wording of Assumption 4, so that the logging requirement does not hold if the Shop initially already contains orders. However, this is not a design flaw.

G. Threats to validity

1) Conclusion validity: An obvious threat to the validity of the conclusions of the case study is the small number of participants. Nevertheless, the different results corresponded sufficiently for them to be seen as a positive first step on the road to a larger scale experiment.

2) Internal validity: Both participants have a similar level of experience with software architecture and requirements engineering, and are familiar with state-of-the-art design processes. While we acknowledge that it would be interesting to investigate the impact of prior architectural experience on the methodology, this was not the goal of this case study. Furthermore, we believe that the experience of the participants adds weight to their answers on the questionnaire. Similarly, not all participants had equal experience with

[1]These implicit assumptions would have been made explicit by applying so-called relaxation rules [1]. However, the subjects were not tasked with applying these due to time constraints.

Table I
CHECKLIST TO ASSESS THE CORRECTNESS OF THE COMPOSITIONS.

Assumption	*A*	*B*	*Ref.*
1. Only one Shop.	✓	✓	✓
This assumptions was provided to both participants in the warm-up exercise, in order to scope the analysis better.			
2. Shops use the same Logger.	✓	?	✓
The reference a Shop has to a Logger should be immutable.			
3. Every order has an authorized originator.	✓		
Whenever the Shop has an order, it was placed by someone that was allowed to place that order.			
4. Orders result from PlaceOrder.	!	✓	✓
Whenever the Shop has an order, it is a direct result from executing the PlaceOrder operation. This implies that the Shop can be trusted to only store actual Orders on behalf of customers.			
5. The Logger is always intact.	✓	!	✓
If the Logger detects that it has been tampered with, the security of the Shop is no longer guaranteed.			
6. Logging is instantaneous.	!	✓	✓
Every invocation of the Log-operation is executed immediately, so as to avoid race conditions between writing an Order and reading it back.			
7. AuthenticationEnforcer configured correctly.	✓	✓	✓
Every AuthenticationEnforcer is configured with the initial Account-Password list. This was alternatively split in two statements "the configuration is initially correct" and "the configuration never changes".			
8. Only one AuthenticationEnforcer.	✓	?	✓
This assumptions was alternatively worded as "always use the same AuthenticationEnforcer".			
9. Shop initially empty.		✓	
The Shop initially does not contain Orders.			

modelling in Alloy. In order to mitigate this, apart from the initial presentation and warm-up exercise, participants could also refer to a short list of templates of common expressions of method invocation and return value handling (i.e., a 'cheat sheet'). These templates were not specific to the task at hand.

3) External validity: The main threat to external validity is the reasonably small size of both the application used in the case study and the tasks performed by the subjects. Additional research is required to validate the applicability of this approach to larger architectures, and its applicability by architects with different formal backgrounds. As this case study only applies a subset of the library, care must be taken to generalize results. We have tried to mitigate this last threat by selecting one of the simplest and one of the more complex patterns in the library for the case study.

V. A FORMAL PATTERN LANGUAGE

This section forms the second contribution of this work. It presents the pattern library constructed according to the method of Section III, and discusses trust assumptions that were found by the authors in the refined pattern models. It contains the patterns *Audit Interceptor*, *Authentication Enforcer*, *Authorization Enforcer*, *Secure Logger* and *Secure Pipe* by Steel et al. [3]. We limit ourselves to an informal discussion of the analysis results. The *Secure Logger* and *Authentication Enforcer* are discussed more in-depth, other results are summarized due to space limitations. Both abstract and refined models of these patterns, including the formalized trust assumptions, are available online [5].

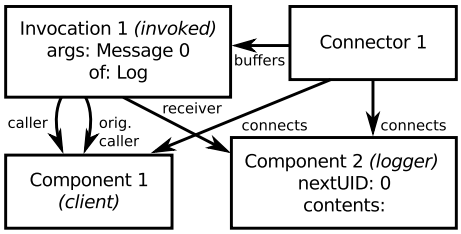

Figure 5. Counterexample illustrating the need for message delivery guarantees.

A. Secure Logger

The Secure Logger is modelled as described in Section III, which in turn is an operationalization of part[2] of its documentation in [3]. The pattern is verified to uphold the security requirement that whenever the Log operation is invoked on a Logger with a specific message, there is a time in the future so that for all further times either that message can be read back from the Logger, or the Verify operation of the Logger returns false.

When verifying that security requirement, counterexamples such as in Figure 5 are generated by the analyzer. This example shows a Component 'client' connected to a Component 'logger'. The client invokes the Log operation on the logger. However, as the Secure Logger does not provide delivery guarantees 'out of the box', it is possible that invocations are never executed (i.e., they are lost). This seemingly trivial counterexample hints at an important trust assumption—if the client can not trust that Log invocations are actually executed (e.g., because of a network problem or an attacker staging a denial of service attack), it is possible that Messages never get logged, effectively deleting them.

This trust assumption could be resolved by equipping the Secure Logger with some form of message delivery notifications, or by ensuring that the middleware offers message delivery guarantees. Ideally, the original pattern description should either document the most common solutions as implementation alternatives, or explicitly document this trust assumption as a relevant security consideration. However, the documentation does not contain this information, which impairs the intended security assurance of the pattern.

The Secure Logger trust assumptions fall in three categories. First, the Secure Logger itself should be trustworthy. This entails that it should not overwrite previous entries, and that the integrity of the nextUID message counter should be protected. Second, the Secure Logger should be properly integrated in the architecture. Its connectors should be reliable (i.e., do not disconnect previously connected components), tamper proof (i.e., provide data integrity), properly initialized (i.e., not be used before the client and the SecureLogger are connected) and provide guaranteed delivery

[2]The original Secure Logger also provides data confidentiality. However, this was not required for accountability and therefore not modelled.

of invocations of the Log operation (i.e., as discussed previously). Additionally, only these secure connectors should be used. Third, as the Secure Logger makes use of cryptographic primitives to digitally sign its messages, it also contains the assumption that the digital signatures should be impossible to forge. Depending on how digital signatures are realized, refining this assumption will in turn result in trust assumptions on key management. Note that the trust assumptions on the connector being tamper proof can be resolved by the Secure Pipe pattern, as documented in Section V-E.

B. Authentication Enforcer

The Authentication Enforcer creates a centralized service to provide authentication of users while encapsulating the details of the authentication mechanism. It does this by creating a Subject which represents a Component executing on behalf of a User. A User authenticates himself by means of a UserID and Credential. Valid User→Credential combinations are stored in a UserStore. The Authentication Enforcer realizes the security requirement that at all times, if a calling component obtains a valid session by executing the Authenticate method with a specific user id and credential, then that session associates the user corresponding to that user id with the calling component.

The elicited assumptions are in three categories. The first category contains general residual goals on authentication, i.e., UserIDs should uniquely represent Users, and clients of the authentication service should only use the UserID and Credential as specified by the User. While the latter assumption might appear odd at first glance, it signifies well-known situations such as forgetting to log out on a public computer. The second category comprises expectations and residual goals on the UserStore—every Authentication Enforcer should have a reference to an effective UserStore, and the User should know the Credentials as contained in the UserStore. The third category contains expectations on the underlying middleware. It assumes that authentication Invocations are instantaneous (i.e., they are executed immediately) in order to avoid race conditions, and it assumes that the components involved in authentication are themselves authenticated on the network layer.

This last assumption hints at the inherent recursiveness of authentication—in order to trust authentication, the Authentication Enforcer should ascertain that the client is authentic. Consider for instance the counterexample in Figure 6. It shows a Component 'client' that invokes the Authenticate operation of the Authentication Enforcer, providing a valid user ID and credential. However, the caller of that invocation is spoofed by a Component 'attacker'. This results in the Authentication Enforcer creating a valid Subject that authenticates the attacker for the specified User, instead of the client. The assumption can be mitigated by either refining the model to authenticate components involved in authentication

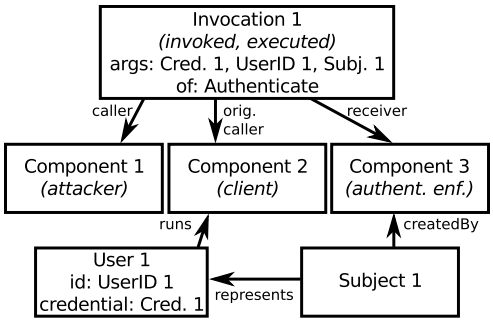

Figure 6. Counterexample illustrating an authentication spoofing attack.

on a lower level (potentially down to the hardware level, as with a tamper proof module), or by applying the Secure Pipe pattern from Section V-E.

C. Audit Interceptor

The Audit Interceptor pattern is used to centralize auditing functionality and support declaratively defining audit events. It realizes the security requirement that for all executed Invocations deemed to be sensitive, there exists a corresponding AuditEvent and AuditInterceptor from which that AuditEvent can be read back in the future.

The trust assumptions of the Audit Interceptor can be subdivided in two categories. First, the AuditInterceptor is assumed to be initialized with the correct configuration, and it should be impossible to tamper with that configuration. Second, every Invocation should be intercepted by at least one AuditInterceptor (i.e., full mediation). Every AuditInterceptor should have an effective Logger, and always read and write to that same Logger. Furthermore, that Logger should be intact (i.e., its contents not deleted). Finally, Invocations to read back AuditEvents are routed to the correct AuditInterceptor. Note that the trust assumptions on the Logger role can be resolved by the Secure Logger pattern from Section V-A.

D. Authorization Enforcer

The Authorization Enforcer acts as a mediator that encapsulates and performs authorization checks. It realizes the following requirement. For all Components that invoke an Operation to which access is controlled, if that Invocation is actually executed, either it was called on the AuthorizationEnforcer (which acts as a mediator), or the Invocation is authorized.

The Authorization Enforcer depends on a correct initialization of its configuration and corresponding protection of its integrity. Similarly, it also depends on full mediation of access to the protected operations, and should be trusted to only forward requests on the behalf of other parties. As was the case with the Secure Logger, this represents the added trust required from components that fulfil a security role. The final assumption denotes that authorization depends

on successful authentication. That last assumption could be fulfilled by the Authentication Enforcer or Secure Pipe.

E. Secure Pipe

The Secure Pipe pattern protects the confidentiality and integrity of communication (i.e., Invocations) between two connected Endpoints. It realizes the requirement that if some data is received and decrypted successfully by an Endpoint, then that data was encrypted by another Endpoint to which the first Endpoint is connected, or both Endpoints are the same.

First, as the Secure Pipe uses cryptographic primitives to realize its requirement, it includes assumptions on those primitives as well. Endpoints should only use the cryptographic key that they 'know' (i.e., have in memory), and only connected Endpoints should share that key. Additionally, the key should be fresh (i.e., not used previously). Second, the Secure Pipe should be trusted to only transmit what it receives from its client (i.e., it should not encrypt and send messages that did not originate from the client). Finally, communication between Endpoints should be authenticated.

As with the Authentication Enforcer, the Secure Pipe abstracts that certain trust relationships are implicitly recursive and can only be solved by depending on the trustworthiness of a lower layer. This results in either an unmitigated trust assumption (e.g., the architect blindly accepts the security of an underlying library), or can be solved by refining the architecture until the 'chain of trust' is grounded in a sufficiently trustworthy source (e.g., by depending on a tamper proof module realized in hardware).

F. Summary

Figure 7 contains the results of cross checking the explicit assumptions with the original pattern documentation, and shows that the majority of the trust assumptions that we found are not documented at all in the original descriptions. This approach can be systematically applied on the whole corpus of security patterns to improve their documentation.

Practically every trust assumptions in the original documentation is related to the use of cryptographic primitives. As the Secure Logger and Secure Pipe patterns depend heavily on cryptography, their documentation is more complete. However, while cryptography trust assumptions are important, they are far from the only trust assumptions, as this section has illustrated.

VI. Discussion

While well-documented security patterns such as those in [3] include reference code to aid the developer in implementing the patterns securely, we conclude that the same patterns contain few hints for the architect to verify that an architecture built around these patterns is secure. This work shows that it is possible to produce reusable formal models that are of practical value to the software architect. The abstract pattern models can be used by the architect as 'reference code' for formal verification purposes. Additionally, the refined pattern models can be used to more exhaustively document the trust assumptions on which the correct operation of the pattern is based. We propose that adding abstract models and explicit trust assumptions to the security pattern description would increase the assurance it offers to the architect.

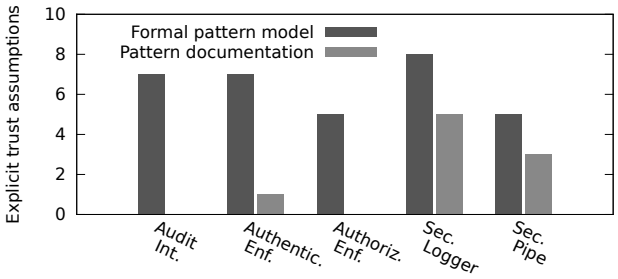

Figure 7. The amount of trust assumptions found per pattern, and the amount of those mentioned in the original pattern documentation.

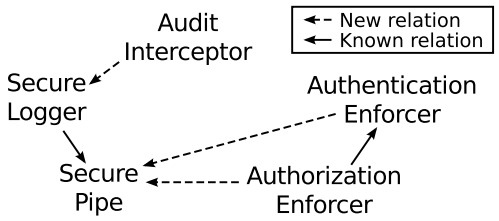

Figure 8. Relations uncovered in the pattern language. The 'known' relations are documented in [3], and were also found by our approach.

Some security patterns depend on trust assumptions that are realized by other patterns. These inter-pattern relations are at the core of pattern languages. The relationships that emerged in the accountability language are shown in Figure 8. They indicate that some security patterns naturally benefit from another. This is corobborated by the experience of the authors in various projects ranging from e-government to e-health and e-publishing, in which these patterns frequently occurred together. While some of these relations are also documented in the original description (i.e., the solid arrows in Figure 8, as mentioned per pattern in the 'related patterns' section [3]), many of them are not (the dashed arrows). Again, adding this information to the pattern documentation could assist the architect in selecting the right patterns for the task at hand. This approach can be used to systematically elicit these inter-pattern relations.

VII. Related work

Many approaches to formally model software systems exist. Assume-guarantee-based reasoning is a family of generic modelling approaches that explicitly take assumptions on the environment of the model into account. An excellent overview on these methods is presented by De Roever et al. [6]. One of those approaches in particular,

lazy compositional verification by Shankar et al. [7], uses a similarly weak characterization of the environment of the component for verification. However, these approaches are not tailored to the software architect. A related architectural modelling approach is the software specification and analysis method (SAM) [8], which allows verification of architectural properties, but does not help in eliciting assumptions. France et al. [9] describe a formal UML-based approach to modelling design patterns. Eckhardt et al. [10] apply model checking to two security patterns for denial of service, and show that their composition leads to a new, improved pattern.

Schumacher et al. [11] propose an ontological approach to modelling security patterns that focuses mainly on finding the right pattern, not on evaluating the contents of individual patterns. Similarly, Sarmah et al. [12] propose a lattice-based classification founded on formal concept analysis using a trust-based security model. Kubo et al. [13] present a methodology to automatically extract security pattern relationships from HTML documents.

Concerning the assessment of the quality of security patterns, Halkindis et al. perform a qualitative analysis of patterns contained in the Blakley and Heath inventory [14]. Konrad et al. apply the same evaluation criteria to the Yoder and Barcalow inventory [15]. To the best of our knowledge, no previous work on finding undocumented assumptions in security patterns exists.

VIII. Conclusion

We have presented a novel approach to model and verify software architectures based on reusable formal models of security patterns. Every pattern is modelled on an abstract level to verify compositional properties and on a refined level to elicit trust assumptions. Non-specialists can successfully use the abstract models to design architectures with reasonable overhead, as validated by a small but rigorous observational study. Work on providing additional tool support for modelling is ongoing.

Additionally, the refined models uncover relevant trust assumptions as illustrated by modelling a pattern language for accountability, which uncovered many previously undocumented issues. The abstract models can be used by practising software architects as reference code to efficiently verify architectures based on these patterns, and the trust assumptions as a checklist to perform more efficient security evaluations. Future work includes a formal framework to decompose generic architectural models and additionally increase scalability.

Acknowledgment

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund KU Leuven. This research is partially funded by the EU FP7 project NESSoS.

References

[1] T. Heyman, R. Scandariato, and W. Joosen, "Security in context: analysis and refinement of software architectures," in *Annual IEEE Computer Software and Applications Conference*, July 2010.

[2] D. Jackson, *Software Abstractions: logic, language and analysis*. The MIT Press, 2006.

[3] C. Steel, R. Nagappan, and R. Lai, *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall, 2005.

[4] T. Heyman, K. Yskout, R. Scandariato, H. Schmidt, and Y. Yu, "The security twin peaks," in *International Symposium on Engineering Secure Software and Systems*, February 2011.

[5] Formal pattern library. [Online]. Available: http://distrinet.cs.kuleuven.be/software/samodels/

[6] W. Roever, *Concurrency verification: introduction to compositional and noncompositional methods*, ser. Cambridge tracts in theoretical computer science. Cambridge University Press, 2001.

[7] N. Shankar, "Lazy compositional verication," in *Compositionality: The Significant Difference*, ser. Lecture Notes in Computer Science, vol. 1536. Springer Berlin Heidelberg, 1998.

[8] X. He, H. Yu, T. Shi, J. Ding, and Y. Deng, "Formally analyzing software architectural specifications using SAM," *The Journal of Systems & Software*, vol. 71, no. 1-2, 2004.

[9] R. France, D. Kim, S. Ghosh, and E. Song, "A UML-based pattern specification technique," *IEEE Transactions on Software Engineering*, vol. 30, no. 3, 2004.

[10] J. Eckhardt, T. Mühlbauer, M. AlTurki, J. Meseguer, and M. Wirsing, "Stable availability under denial of service attacks through formal patterns," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7212.

[11] M. Schumacher, "A theoretical model for security patterns," in *Security Engineering with Patterns*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, vol. 2754, ch. 8.

[12] A. Sarmah, S. Hazarika, and S. Sinha, "Security pattern lattice: A formal model to organize security patterns," in *19th International Workshop on Database and Expert Systems Application (DEXA '08)*, 9 2008.

[13] A. Kubo, H. Washizaki, and Y. Fukazawa, "Extracting relations among security patterns," in *International Workshop on Software Patterns and Quality (SPAQu)*, 2007.

[14] S. T. Halkidis, A. Chatzigeorgiou, and G. Stephanides, "A qualitative evaluation of security patterns," in *International Conference on Information and Communications Security (ICICS)*, Malaga, Spain, October 2004.

[15] S. Konrad, B. H. Cheng, and L. A. Campbell, "Using security patterns to model and analyze security requirements," in *IEEE International Conference on Requirements Engineering (RE)*, Monterey Bay, CA, USA, 2003.