A Rigorous Methodology for Security Architecture Modeling and Verification

Yomna Ali
American University in Cairo
yomnas@aucegypt.edu

Sherif El-Kassas
American University in Cairo
sherif@aucegypt.edu

Mohy Mahmoud
American University in Cairo
mohym@aucegypt.edu

Abstract

This paper introduces a rigorous methodology for utilizing threat modeling in building secure software architectures using SAM (Software Architecture Modeling framework) and verifying them formally using Symbolic Model Checking. Security mitigations are expressed as constraints over a high-level SAM model and are used to refine it into a secure constrained model. We also, propose a translation from SAM Secure models into the SMV model checker where the threats and the elicited security properties from the threat modeling process are used as inputs to the verification phase as well. This method is developed with the aim of bridging the gap between informal security requirements and their formal representation and verification.

1. Introduction

There is a lack of a well-defined process for the analysis, architecture and design of secure software systems [1]. At the same time, the lack of formality in defining security requirements makes it harder to verify that they are met by the underlying security architecture. In this paper, we propose a rigorous methodology for building and verifying secure system architectures guided by the process of threat modeling that is performed with both architecture and verification in mind. Security constraints are expressed as mitigations and are used as suggested by [2] in refining a high-level architecture into a secure architecture model. The resulting secure model is translated to a model checker for verification and output from the threat model is also utilized in verifying that the security properties elicited from the threat model are satisfied by the architecture.

2. Related Work

Our research intercepts with several areas of security engineering, all aimed at the early integration of security in the software life cycle; they are: Adversary Modeling, Architecture Modeling and Architecture Verification. In Adversary modeling, several methods have been devised in order to model the goals of the adversary and the means he will use to achieve these goals. Attack trees [3], goal-oriented analysis [4], and misuse cases [5] are all forms of adversary modeling, targeting the elicitation of security requirements. Threat modeling, also a form of adversary modeling and utilized by Myagmar et al. [6], and Howard and LeBlanc [7] presents a systematic way of determining the threats to a software system. Threat modeling starts with an initial system decomposition that could be achieved using data flow or UML activity diagrams. This decomposition is useful in identifying assets that need to be protected, their access points, and their vulnerabilities. After stepping through each asset, and identifying possible threats as suggested by Xu and Nygard [1], Howard and LeBlanc [7], Myagmar et al. [6], using the STRIDE threat categories [7], threats could be documented in a variety of ways (like threat trees, or threat outlines [3]). Threat modeling has a wider-scope than the attack modeling techniques that consider only the attacker's perspective because system-specific threats require deeper analysis of the unique qualities of the system being modeled [6].

Most of the above mentioned adversary modeling techniques lack two aspects: (1) they leave it open as to how this modeling process will affect the choice of architecture components, or how they could be captured in the architecture and design phases when as Haley et al. [8] indicate: the elaboration of requirements and architecture should proceed in parallel, each influencing the other (2) they mostly lack a formal means by which these security properties could be modeled, traced, and verified in the resulting secure architecture.

Xu et al address the first concern by suggesting utilizing misuse cases in guiding the choice of architecture components. Use/misuse case diagrams are used for deciding on candidate components. They use a tabular approach of components in the rows and the use/misuse cases utilizing them in the columns, to decide on the services that the system must provide and prevent the ways that the system might be misused [9]. Then, using a fish-eye diagram of the candidate architecture, the components that need change are highlighted in size or in color according to the number of interactions that each component has

978-0-7695-3450-3/09 $25.00 © 2009 IEEE

from the table. This can help system designers later decide on the best fitting candidate architecture. However, the lack of formality in their approach would make it harder to trace and verify the security requirements in their resulting architecture.

Lamsweerde et al [10] and Lamsweerde [11], address the second concern, as they use temporal logic for formally specifying their security goals. Anti-models are introduced in [11] as a way of defining anti-goals as opposed to system goals. Anti-goals are derived using obstacle analysis extended to handle malicious obstacles building on the KAOS framework for generating and resolving obstacles to requirements achievement. Anti-goals are derived through goal negation of the CIA patterns, which are in turn instantiated to objects of the object model. Through progressive refinements using techniques such as goal negation, terminal anti-goals can be derived, which can be used to identify anti-requirements and vulnerabilities. Later, these anti-goal trees are used to build the object and agent anti-models and at the end, alternative countermeasures are derived to augment the primal model of security goals [11]. Lamsweerde [10, 11] address the importance of exception handling at the requirements level and therefore maintain their analysis at the goal level. In our research, we also utilize linear time temporal logic for security property specification; however through the use of a new threat modeling template, with specific architecture additions and utilizing formal definitions, we continue to follow the requirements from analysis to initial design and its verification to make sure that we have a process for continually verifying and refining the architecture model using security property specifications. In our proposed methodology, the outcome of threat modeling is used to both influence the architecture and verification phases, and at the same time, the outcome of the verification phase can reveal new threats from flawed design uncovered by the tests, thereby refining both the threat and architecture models accordingly.

For architecture modeling, Xu and Nygard [1], propose an approach for modeling threats as Petri nets and mitigations as aspect-oriented Petri nets, thereby providing a step towards the formal specification and verification of security requirements and mitigations. SAM, the Software Architecture Modeling framework, can also be used as suggested by Deng et al. [2] to provide a multi-level architectural model with dual notation (Petri nets and temporal logic) for "describing different aspects of architecture level design such as structure, behavior and constraints". Fu et al. [12] propose a variation of the SAM framework called SO-SAM: a service-oriented software architecture model and an extension of the SAM model specifically for modeling web services applications. Deng et al. [2] propose that desired security properties should be expressed as constraints or policies over the SAM high-level architecture using temporal logic. These constraints are then used to further refine the software architecture by decomposing [2] the high-level architectural model.

As for verification, and given a petri-net based architecture model, both Xu and Nygard [1], and Deng et al. [2] suggest reachability analysis for security architecture verification. Whereas Fu et al. [12] suggest translating the architecture models into one of the model checking languages, and presenting the checker with logical properties (such as liveness and deadlock freedom) to verify that they are satisfied by the translated models. Unlike reachability analysis, model checking does not require the generation of all states before properties can be examined. Therefore it does not suffer from the state explosion problem [13].

In order to verify an architecture model using a model checker, a mapping between the architecture and the model checker language has to be defined. Tanuan [14] suggests utilizing model checking in verifying the constraints in the UML specifications, where a specific mapping between UML and SMV is proposed to verify UML specifications. At the same time, others suggest the mapping of SAM petri net behavior models and property specifications into SMV. For example, He et al. [15] propose utilizing symbolic model checking for verifying correctness of an architecture specification in SAM. They suggest translation guidelines for translating petri net behavior models into SMV and then test CTL properties against the translated transition system for verifying the architecture specification. He et al. [16] also propose a translation procedure of SAM behavior models into SMV.

Most of the above proposed methods for earlier security integration usually target either the phases of requirement specification or architecture modeling but not both, failing to formally trace security requirements from the requirement analysis phase to architecture, design, and verification phases. At the same time, there is a lack of a rigorous translation methodology specifically targeted for translating the SAM models into SMV and a lack of a verification method targeted at verifying security properties in the presence of threats using symbolic model checking where most research is targeted at verifying correctness and safety properties. In this paper we show that we can express security properties formally using temporal logic over an SAM architecture

model, as well as propose a translation methodology between the SAM architecture model and its security properties and the SMV model checker, for verifying that the security properties are met by the architecture.

3. Methodology

The methodology presented in this paper builds on the work of Xu and Nygard [1], Deng et al. [2], Fu et al. [12], He et al. [16], and Myagmar et al. [6]. The goal of this research is to guide the process of secure architecture modeling and verification by performing threat modeling [6, 7] earlier in the software life cycle during the requirements analysis phase. We propose a new threat modeling template that allows the designer to specify architectural artifacts and constraints as mitigations along side the security solutions of the classical threat models [7]. The template also allows the designer to specify the security property specifications that the resulting system should meet if the threat is truly mitigated. For architecture modeling, the threat model's architectural mitigations and constraints could be used to refine a high-level architecture model into secure behavior models. Whereas for architecture verification, the threat model's logical property specifications as well as the threat descriptions could be used as inputs to the verification phase to verify that the threats do not violate the security properties under the provided secure architecture model. This way, threat modeling could rigorously influence both secure architecture modeling and verification.

Threat models give threat descriptions, and provide security mitigations as well as elicit security property specifications. In order to bridge the gap between informal threat descriptions and formal architecture modeling, we propose formally specifying mitigation constraints in temporal logic over the high level Service-Oriented Software Architecture Model (SO-SAM) proposed by Fu et al. [12]. These constraints could then formally guide secure architecture decomposition (as proposed by Deng et al. [2]).

The resulting architecture model, represented by SAM Petri net behavior models is verified as Fu et al [12], and He et al. [16] suggest by translating it into a high-level model checking language (where it could be checked for the absence of threats by verifying the stated security properties). We suggest for this purpose a translation methodology between the SAM Architecture model and the SMV model checker. Counter examples provided by the model checker can help the designer identify the problem and refine both the threat and architecture models accordingly.

The proposed methodology like that suggested by Hall et al. [17], attempts to build correctness into every step by suggesting a rigorous requirements definition, formal architecture modeling using the Software Architecture Modeling framework (SAM), and formal verification of the resulting architecture against the security property specifications using model checking. Figure 1, shows the steps of the proposed methodology:

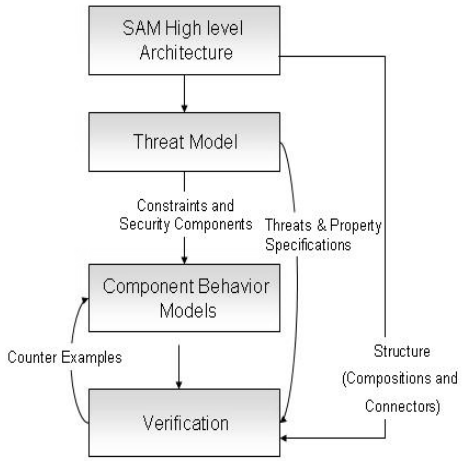

Figure 1. Proposed Methodology

Step 1: High-level Architecture:
Build a high-level architecture of the system. Figure 2, shows the e-company server side only of a high level architecture of a shopping cart application proposed by Fu et al. [12], composed of the Warehouse and Order components and their interface input/output ports (petri net nodes responsible for communication with other components or system parts). Components are numbered and petri net ports are concatenated with numbers denoting which components produce/consume their tokens. For example request_7 is an input interface port to Warehouse component (number 7) that receives a request token to be consumed by the component; whereas response_72 is an output interface port that sends a response token from warehouse component to another component numbered 2 in the model (not shown in figure).

Step 2: Threat Modeling:
Perform threat modeling on the given architecture, where threats to assets and access points are identified and classified based on the STRIDE categories. Fill the proposed threat modeling template with the security threats and their respective mitigations. This template enhances the classical model [7] -- where security techniques (such as authentication, authorization, encryption, etc.) are

specified as mitigations -- with the ability to specify architectural mitigations, logical constraints, and property specifications. These enhancements utilize a combination of formal notation, using first order temporal logic, and informal English descriptions.

Step 3: Building Secure Behavior Models:

Translate each mitigation to an equivalent Petri net model [2], to build a component behavior model using the new transitions and the new places suggested by the mitigation, and the constraints imposed over existing or new transitions.

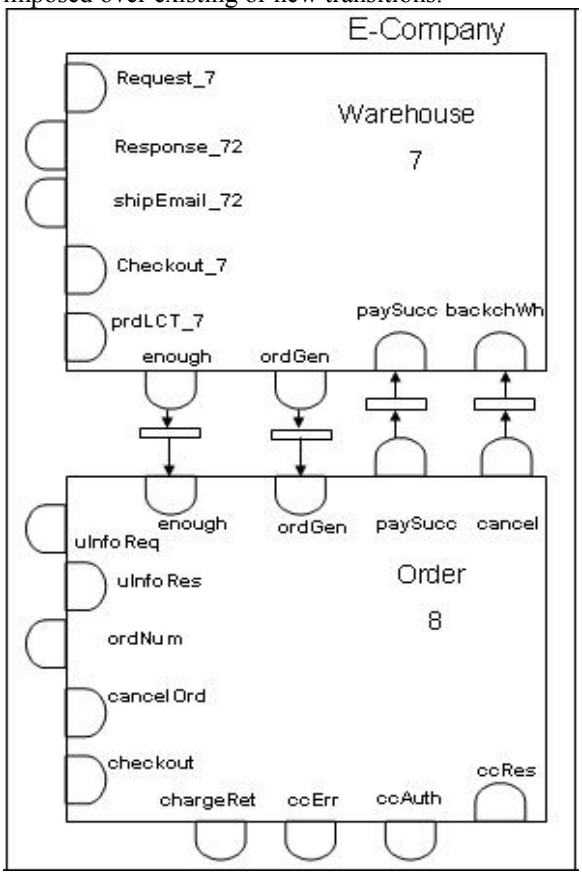

Figure modified from [12]

Figure 2. High level SAM Architecture of E-Company

Step 4: Building the Constraint Architecture Model (CRM):

- Build a constraint architecture model for the entire architecture by plugging in the component behavior models at their proper places in the high-level architecture.
- For each new security technique specified as a mitigation, insert a black-box new component in the detailed architecture at the proper location
- Use system wide constraints to link the new components to existing components according to the provided policy constraints. These system wide constraints are too realized using new transitions and places.
- For each mitigation realized by a new component: if the component is a black box off-the-shelf security solution, we insert a place-holder for the component with its interface ports without refining its internal structure. Otherwise, we refine it by applying threat modeling to it (go back to step 2).

Step 5: Verification:

- Start translating the SAM model into SMV to perform model checking:
- Use the SAM high-level architecture to provide the structure for the SMV model by dividing it into system, composition, and component modules, as well as connectors between components.
- Translate each component behavior model into a component module using our suggested translation methodology
- Translate the LTL security properties specified by the threat model into CTL and insert them into the SMV model
- Transform each threat into a set of parameters that change the initial marking of the model.
- Check to see if the security properties are satisfied by the given model, to determine if the architecture is secure given the corresponding threats or not.

Step 6: Refinement:

Use counter examples provided by the model checker to refine the threat and architecture models.

3.1. Threat Modeling the SAM Architecture Model

The proposed threat model integrates the classical form of threat models where security techniques are specified as mitigations, with a new model where architectural constraints, components, or artifacts are specified as mitigations as well. These architectural mitigations are used to guide the refinement of the SAM model into secure constraint behavior models. The architectural nature of our threat model facilitates integrating application-security threats resulting from implementation flaws along side the classical security threats resulting from compromising the confidentiality, integrity or authenticity of the system as a whole. A classical threat modeling template would at minimum include a threat Title, a threat Category according to STRIDE model, and the mitigation technique (access control, encryption, etc). The new threat modeling template enhances the classical model with the following architecture and verification-related additions:

- **Threat Type:** threats could be either:

- **Component Threats:** These are threats that are specific to a component, examples are business logic threats such as price or quantity tampering.
- **Composition Threats:** These are threats that involve more than one component in the same composition. Examples are threats resulting from a flawed communication between components inside a composition, causing for example a validation bypass.
- **System-wide Threats:** These are threats that apply to an entire system (i.e. the output of the system as a whole will be affected by such a threat); for example a threat to the whole web service with all its respective components, like a spoofing identity threat, or an elevation of privilege threat.

- **Threat Target:** Name of the architectural artifact that the threat could apply to, it could be the name of an entry/exit point of a component, a composition, or the entire system.
- **Mitigations:** Mitigations could be either:
  - **A Logical Constraint**
    - **An intermediate component constraint**: This could be realized by a logic constraint enforced over an existing transition in the High level Petri net or by adding a new transition/place to the architecture and applying this constraint to it.
    - **A system wide constraint** – enforces a policy governing interactions between components: expressed as a constraint over one or more of the system components, or the entire system.
  - **An Architectural Mitigation**: An architectural mitigation enforces a certain control flow. For example, what places should be enablers of a certain transition.
  - **A security technique:** A security technique could be any of the standard techniques used to provide for security, examples are: firewalls, content inspection, access control, or encryption modules.
- **Location/Realized By:** This entry should specify where the mitigation should be applied: either inside the component (if a constraint), or between components (if a system-wide constraint). At the same time, it could specify how it will be realized, by either imposing the constraint over an existing transition, or by adding a new transition to the component's internal architecture, in which case we should specify what places are in its preset (enablers). If the mitigation is a new component, the designer should specify its interface input and output ports, as well as what components it will interface with.
- **Security property specification:** If the mitigation is a logical constraint, it could be expressed inside a security property specification using LTL (Linear Time Temporal Logic). This property could be later used to verify the absence of the threat. LTL describes how the state of the world evolves over time thereby focusing on the ordering rather than the exact timing of events. Temporal formulas are constructed from predicate symbols (equality and propositions), function symbols, constants, variables, the logical operators (¬, ∧, ∨, ⊃, ∈ and ≡), the quantifiers (∃ and ∀), and the temporal operators (□: Always, ◊: Eventually, and O: Next). [2]

Table 1, shows an example of a component application threat, which is the threat that a customer purchases products at a lower price. This threat, also suggested by Xu and Nygard [1], lies in the possibility of a customer changing the price of the purchased product to reduce the amount of payment he has to make. The type of mitigation is two fold:

- Add a ***component constraint*** that is a constraint on the internal behavior of the component. For example, to be able to mitigate this sort of attack, we need to compare the prices sent in the purchased products' list (at port prdLCT) with the prices stored in the warehouse's product database.
- An ***Architectural mitigation*** (enforces a certain control flow): Price validation should not occur unless we have validated that the request is a valid checkout request (at port checkout).

The component property specification is derived, by using only interface input/output ports of the component and the new component constraint, and it indicates that when we have a checkout request and a valid price, ordGen_78 should eventually be enabled.

3.2. Decomposing the SAM Architecture Model

Given the high level architecture and the new transitions and places/nodes over which mitigating constraints are enforced, we could start decomposing the SO-SAM high level architecture to the corresponding constraint architecture models. A behavior model of each component results from refining its high level architecture using the threat model mitigations augmented with the functional behavior model. This could result in a behavior model for each component.

Starting from the input interface ports of each component in the SO-SAM model (Figure 2), we use the architectural mitigations and constraints of the threat model, to refine the high-level architecture.

Table 1. Component Threat Example

Threat /Description/Type	Threat Target	Mitigation	Location	Security Property Specification
Component Threat: Customer purchasing products at a lower price **Description:** A customer may change the price of the purchased items in the HTML form at the client side, to pay less than he should pay. **Type:** Tampering	**Warehouse Component**, Entry points: checkout request (checkout_7 , prdLCT_7)	**1-Add component constraint**: Name: valid_price prdLCT.price=PRICES(pname) (i.e. validate that the price in the product list is the same as the one stored in the products DB) **2-Architectural Mitigation:** Price Validation should only be enabled when validCheckout is enabled	**In-Component transition:** Validate_price **New**: Yes **Enabled By:** validCheckout **In-Component Token:** ValidPrice **New:** Yes	**Type:** Component Property When we have a checkout request and a valid price, ordGen_78 should eventually be enabled □ ((prdLCT_7 ∧ checkout_7 ∧ **valid_checkout** ∧**(prdLCT.price=PRICES(pname)))** →◊ordGen_78)

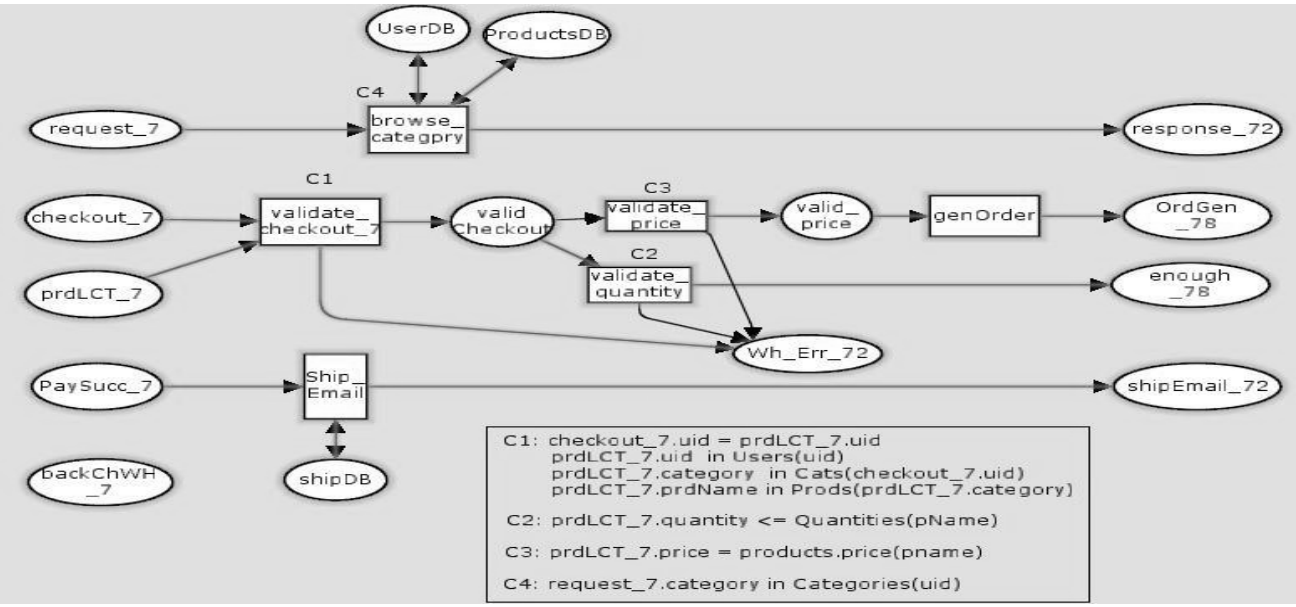

Figure 3. Warehouse-component Behavior Model

Component constraint mitigations are enforced by connecting input interface ports with an internal node or an output interface port using a new transition and imposing the security constraint as an assertion on the new transition as indicated by Deng et al. [2]. This process is repeated until we reach a transition that places a token in one of the interface output ports of the component. Figure 3, shows the resulting behavior model from decomposing the warehouse component of Figure 2. From figure 3, we see that possible entry points to the warehouse component are the input interface ports (that are not enabled by any transition belonging to the component) like request_7, checkout_7 and prdLCT_7, paySucc_7, and backChWh_7, while exit points are the output interface ports that do not fire any transition belonging to the component like response_72, ordGen_78, enough_78, and shipEmail_72. This figure was constructed by starting with the input interface ports of the warehouse component in the high-level architecture (Figure 2) then given the suggested constraints and architectural mitigations of the threat model, we start building the inner workings of the component. For example, if we want to build the control flow starting from entry point checkout_7:

- We locate the threats in which checkout_7 is mentioned in the *Target* column (see Table 1)
- Then we add the specified mitigation constraint(s), for example:
  {prdLCT_7.uid} ∩ USERS(uid) ∧
  prdLCT_7.category ∈ CATEGORIES(uid) ∧
  prdLCT_7.prdName∈ PRODUCTS(category)
  (i.e. Check for correct user id, correct category and valid product) by enforcing it on the specified *In-component-transition* validate_checkout.
- Then we connect the new transition to the place/node specified by the *In-Component token*, which in our case was valid_checkout.

Afterwards, we find another threat targeting the checkout process (ex Table 1), and we add the specified mitigation constraints:

(prdLCT.price = PRICES(pname) and prdLCT.quantity <= QUANTITIES (pname) ∧ prdLCT.quantity > 0) to the suggested transitions validate_price and validate_quantity respectively. The architectural mitigation specifies that these transitions should not be fired unless checkout was valid, and are hence enabled by the availability of a token in the place validCheckout (mandating a certain control flow). Finally, we connect the new transition(s) to the places specified by the *In-Component token*, which in our case were ordGen_78 and enough_78 respectively. backChWH_7 is an input port that receives an input token from Order Component to cancel warehouse order operations, but that does not result in the warehouse component producing output to the outside world. Therefore, it is not connected to any output port in the figure.

3.3. Verification

SAM architecture is hierarchical and is defined as a set of compositions; each composition consists of a set of components, connectors and constrains. At the lowest level, each element whether a component or a connector is defined using a behavior model and a set of temporal logic property specifications. If we can find a mapping between the afore-mentioned SAM elements and the SMV constructs, then we can arguably translate different SAM architecture models into SMV model checking programs to verify that they meet their stated security properties.

Most of the work targeted at architecture verification, focuses on verifying liveness and correctness properties. At the same time, it mostly focuses on translating the detailed behavioral models of individual components to SMV [15], [16] without regard to the interaction between components and how they behave in the entire system. In addition to translating the lower level abstraction we pay specific regard to modeling compositions, Commercial off the shelf products (COTS), how components interact in the system, and the security constraints that apply to the architecture as a whole. Therefore, and in order to preserve structural properties along side behavioral ones, we provide a translation of the SAM model (i.e. the high-level decomposition into compositions and their constituent components, and connectors) into SMV. SAM elements are textually described clearly in [16]. Our translation has three main steps:

3.3.1. Translating SAM Petri Net Behavior Models. An SMV program is made up of the sections: VAR, INIT, ASSIGN, DEFINE and SPEC [18]. A general procedure for translating SAM behavior models into SMV is suggested by He et al. [15]. They suggest the mapping of every place to a corresponding boolean variable under the VAR section, every enabling condition of a transition to a symbol represented by an expression in the DEFINE section, the initial marking to initializations in the INIT section, and the specifications (in our case the security property specifications), into CTL formulas in the SPEC section of the SMV program. We will base our petri net translation method over this general translation with additions specific to the translation of SAM high-level security architectures. For example, we specify each security constraint as an expression defined in the DEFINE section, and use this constraint to decide on the next value of output places when the corresponding transition fires, hence enforcing the execution of the security constraint. We prefer to define the next values of the Petri net nodes using ASSIGN rather than TRANS because of logical absurdities that can occur in TRANS declarations [18]. The next state values of places are defined using a case expression under the ASSIGN keyword [15], where setting the value of the variable is equivalent to placing a token in its corresponding place, and resetting it, is equivalent to consuming this token, otherwise the value is kept unchanged this cycle.

3.3.2. Translating SAM Components. SMV gives the user a chance to define modules, where each module can be passed parameters, and hold its own set of local variables and definitions. This SMV element maps to the definition of a component in the high level SAM architecture. A module helps abstract the inner workings of the component, and it need only be passed the input/output variables which represent interface ports (external nodes) in our case. This way the encapsulating module would in a way map to the high-level SAM composition that includes the inner components, where the properties of the composition could be verified with the existence of these components, but without having to concern ourselves with how they actually work. Figure 4, shows a sample translation of a SAM component into SMV. Constraints such as: user1 has to belong to the set of valid user ids, and product1 has to be a valid product id are defined as logical expressions in the DEFINE section and used in the ASSIGN section to decide on the next values of the component's petri net nodes. External in/out nodes' values are formally passed from/to the encapsulating composition responsible for setting/consuming their values.

3.3.3. Translating SAM Compositions. A composition is also defined using a module in SMV; a composition module in SMV could also contain instance variables of other modules representing each of its constituent components. The function of a

composition is to provide communication between inner components as well as communication with the outside world. Interfacing ports/external nodes are defined as variables inside the composition and are passed by reference to their components. Their values are set by the arrival of tokens from other components or the outside world by the forwarding transitions provided by the composition. Each component is represented by a similar declaration to a variable inside the composition module. Each component is instantiated by the composition in the VAR section, and the input and output ports of each component are defined and initialized in the composition's VAR and INIT sections respectively.

3.3.4. Translating SAM Connectors. A connector is a building block that enables interaction among components inside compositions. In our translation, we represent connectors using forwarding transitions that exist in the composition module and provide interaction between its components. Each forwarding transition is represented using an assignment statement that changes the next value of an input port of one of the components using the token generated by the output port of another. This way the composition handles the communication between its different components.

3.3.5. Translating SAM Properties. To verify the correctness of the produced model, we need to test several types of properties: Component properties, Composition properties, and System-wide properties. Security properties are elicited by the threat model and are used in verifying the absence of security threats. SAM property specifications should be transformed from first order LTL (Linear Time Logic) to CTL utilized by SMV, which is a propositional branching-time temporal logic. Whereas LTL considers only one path of computation down a certain state, CTL considers all possible paths from a given state. According to He et al. [16], an LTL formula can be equally expressed in CTL if its execution lies within the common time fragment of LTL and CTL, hence system properties such as liveness and safety can be equally expressed in either computational model without its satisfiability or validity being affected [16]. This can be done by adding a universal path quantifier in front of an LTL formula to transform it to CTL [16], that is an LTL path formula is converted into CTL by quantifying over all the paths using A (universal quantifier denoting all paths).

- **Component Properties**

In component-wide properties, we need to verify the correctness of the component, i.e. that it meets its stated specifications. Dwyer et al. suggest the use of property specification patterns for writing CTL properties [19]. In our model, most of the properties are ones that fall under the "Response" pattern with a global scope. Response property patterns describe cause-effect relationships between two different events, where if the cause occurs, it must be followed by the occurrence of the effect [19]. Component properties in our SMV model would basically look like this:

AG (input1 & input2 & … & constraint -> AF (output 1 & output2 & …)) (1)

Inputs and outputs are strictly input/output interface ports (non-internal nodes) of the component. *Constraint* is a certain test (logical expression) imposed on a transition that when true, a transition that enables the output places should be fired. The following is a translation of the LTL security property in Table 1 into its equivalent CTL form:

AG (checkout_7 & prdLCT_7 & WH.valid_checkout & WH.valid_price -> AF (ordGen_78))

- **Composition Properties**

In composition properties, we need to verify the interaction between components. So any property that involves interface ports starting at one component and interface ports ending at a different component in the most general of terms is considered a composition property. In a composition-wide property inputs are strictly input interface ports of the first component, while outputs are output interface ports of the second component:

AG (COMP1.input1 & COMP1.input2 &..& constraint -> AF (COMP2.output1)) (2)

Note: the component.port notation is used to indicate that Comp1 and Comp2 are variable instances inside the composition and that they are used to access input and output ports of these components/modules.

- **System-wide Properties**

A perimeter component, which could be a commercial-off-the-shelf (COT) product for example, may be a component whose functionality affects the rest of the system, like a firewall for example, where depending on the result of the firewall validation the request is either passed to the respective service or denied. Therefore, COT properties are not only limited to being component-wide properties but they are also system-wide. For example, when a firewall (*FW*) intercepts a web service request (*wsRequest*) and finds that it is valid, it should expect a web service response (*wsResponse*) from the consecutive composition and eventually enable *fwResponse* whether this response holds the requested data or an error. This could be expressed in CTL as follows:

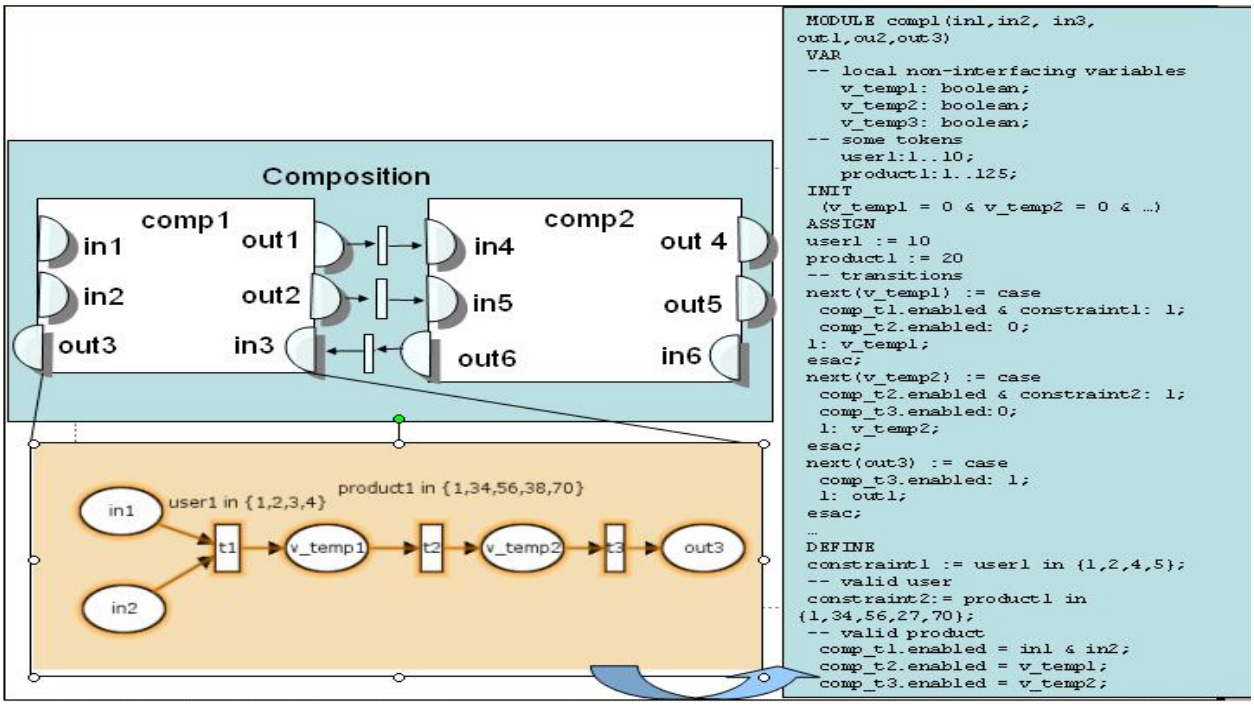

Figure 4. Sample translation of an SAM component to an SMV module

AG (wsRequest & FW.valid_request & FW.valid_data -> AF !(fwResponse = noRes)) (3)

Whereas if you have an invalid request, you should eventually get an invalid fwResponse:

AG(wsRequest & !(FW.valid_request & FW.valid_data) ->AF (fwResponse = invalidRes)) (4)

Note how the truth value of the above properties depends on the behavior of the entire system.

3.4. Refinement

Model checking provides us with the ability to test our architecture behavior models against the security property specifications. SMV automatically executes the model, using the provided marking values to determine the new state of the model at each clock cycle. The goal of refinement is trying to figure out what caused a property violation, then making the necessary changes to the model such that the property is satisfied. In our case, this is achieved by using SMV counter examples to determine what firing sequence caused a property violation in the model. Figure 5, shows the SMV output after executing our translated model, which shows how one of the security properties was violated under the initial design. When tracing back the firing sequence through the SMV trace (lower pane) we discovered that the property (an order is generated only if order info, credit card info, checkout info is valid) would be satisfied under normal conditions however under other threat conditions it would not (ex. when user replays another's valid user information without being requested for his user info). This guided us to an architecture flaw that assumed if we have a valid user information response then the order information must be correct i.e. valid price and quantity (since no user information is requested unless order info was verified). This lead us to add the threat "possible replay of valid user information to checkout an invalid order (ex. with an invalid price)", and to remodel our architecture so as to check on order validity just before generating the order. By repeatedly performing this task we are able to reach a model that satisfies its security properties under both normal and threat conditions.

4. Conclusion

Most of the work proposed for earlier security integration usually targets either the phases of requirement specification or architecture modeling but not both, failing to formally trace security requirements from the requirement analysis phase to the architecture, design, and verification phases. At the same time, no rigorous process was proposed to verify the security properties of architecture models beyond testing for Petri-net correctness properties such as liveness and being deadlock-free. This research proposes a rigorous methodology for the analysis, modeling and verification of secure software architectures guided by the process of threat modeling. By combining threat modeling for

systematic requirement elicitation with the formal representation of security constraints and architectural artifacts (as mitigations using temporal logic), we are able to narrow down the gap between informal requirement specification and formal architecture modeling since this would enable the requirements analysis phase to have a direct impact on architecture decomposition. Moreover, formal architecture modeling using modeling frameworks like SAM facilitates subsequent formal architecture verification. By suggesting a mapping between SAM elements and the SMV model checker, we are able to easily translate the resulting secure architecture models into SMV to verify their stated properties.

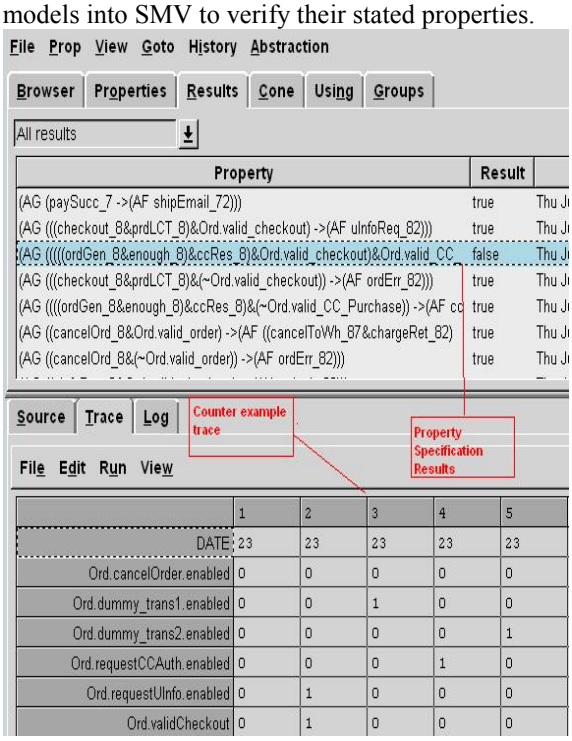

Figure 5. SMV Trace

5. References

[1] Dianxiang Xu, Kendall E. Nygard, "Threat-Driven Modeling and Verification of Secure Software Using Aspect-Oriented Petri Nets," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 265-278, Apr., 2006.

[2] Yi Deng, Jiacun Wang, Jeffrey J.P. Tsai, Konstantin Beznosov, "An Approach for Modeling and Analysis of Security System Architectures," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 5, pp. 1099-1119, Sept/Oct, 2003.

[3] Bruce Schneier, "Attack Trees," *Dr. Dobb's Journal* December 1999. http://www.schneier.com/paper-attacktrees-fig1.html

[4] Ebenezer Oladimeji, Sam Supakkul, and Lawrence Chung, "Security Threat Modeling: A Goal-Oriented Approach," *Proceedings of SEA'06*, Dallas, TX, Dec. 2006

[5] Guttorm Sindre, and Andreas L. Opdahl. "Eliciting security requirements by misuse cases." *Proceedings of TOOLS Pacific 2000*, pp. 120-131, 2000.

[6] Suvda Myagmar, Adam J. Lee, and William Yurcik, "Threat Modeling as a Basis for Security Requirements," *Symposium on Requirements Engineering for Information Security (SREIS)* in conjunction with *13th IEEE International Requirements Engineering Conference (RE)*, Paris, France, Aug., 2005.

[7] Michael Howard and David LeBlanc, "Writing Secure Code," *Microsoft Press*, 2002.

[8] Charles B. Haley, Robin C. Laney, Bashar Nuseibeh, "Deriving Security Requirements from Crosscutting Threat Descriptions," *Proceedings of the Third Int'l Conf. Aspect-Oriented Software Development,* pp. 112-121, 2004.

[9] Dianxiang Xu and Josh Pauli, "Threat-Driven Design and Analysis of Secure Software Architectures," *Journal of Information Assurance (JIAS),* vol. 1, no. 2, June 2006.

[10] Axel van Lamsweerde and Emmanuel Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering," *IEEE Transactions on Software Enineering. Special Issue on Exception Handling,* 2000.

[11] Axel van Lamsweerde, "Elaborating Security Requirements by Construction of Intentional Anti-Models," *Proceedings of ICSE'04, 26th International Conference on Software Engineering,*Edinburgh, pp.148-157, May 2004.

[12] Yujian Fu, Zhijiang Dong, Xudong, "Modeling, Validating and Automating Composition of Web Services," *ACM International Conference Proceeding Series, Proceedings of the 6th international conference on Web engineering*, 2006.

[13] Mihir M. Ayachit and Haiping Xu, "A Petri Net Based XML Firewall Security Model for Web Services Invocation," *Proceedings of the International Conference on Communication, Network, and Information Security (CNIS 2006)*, pp. 61-67, MIT, Cambridge, Massachusetts, USA, Oct, 2006.

[14] Meyer C. Tanuan, "Automated analysis of unified modeling language (UML) specifications." *Master's thesis presented to the University of Waterloo*, August 2001.

[15] Xudong He, Junhua Ding, J. Wang and Yi Deng, "Model checking software architecture specifications in SAM", *Proceedings of International Conference on Software Engineering and Knowledge Engineering*, Ischia, Italy, July 15-19, 2002.

[16] Xudong He, Huiqun Yu, Tianjun Shi, Junhua Ding, Yi Deng, "Formally analyzing software architectural specifications using SAM," *Journal of Systems and Software* 71 (1–2), pp.11–29, 2004.

[17] Anthony Hall and Roderick Chapman, "Correctness by Construction: Developing a. Commercial Secure System," *IEEE Software, Jan/Feb 2002*, pp18 – 25, 2002.

[18] Ken McMillan. "The SMV System". Carnegie-Mellon University, Pittsburgh, PA, Feb.1992.

[19] Matthew B. Dwyer , George S. Avrunin , James C. Corbett. "Property Specification Patterns for Finite-State Verification." *2nd Workshop on Formal Methods in Software Practice,* pp 7 - 15, Clearwater Beach, FL, USA, 1998.