Introducing Security Building Block Models

Andre Rein, Carsten Rudolph
Fraunhofer Institute for Secure Information Technology
Rheinstrasse 75
Darmstadt, Germany
Email: {andre.rein, carsten.rudolph}@sit.fraunhofer.de

Jose Fran. Ruiz, Marcos Arjona
Computer Science Department
University of Malaga
Malaga, Spain
Email: {joseruiz, marcos}@lcc.uma.es

***Abstract*—In today's software development process, security related design decisions are rarely made early in the overall process. Even if security is considered early, this means that in most cases a more-or-less encompassing security requirement analyses is made; Based on this analysis best-practices, ad-hoc design decisions or individual expertise is used to integrate security during the development process or after weaknesses are found after the deployment. This paper introduces Security Building Block Models which are used to build security related components, namely Security Building Blocks. These Security Building Blocks represent concrete security solutions, so called Security Properties, introduced in other publications of the SecFutur project [1]. The goal of this approach is to provide already defined and tested security related software components, which can be used early in the overall development process, to support security-design-decision already while modeling the software-system. The paper shortly describes this new Security Engineering Process with its requirement analysis and definition of Security Properties and how the Security Building Block Model fits into this approach. Additionally the Security Building Block Model is presented in detail. All artifacts and relationships of the model are described. Short examples finish up the paper to show the creation of the Security Building Blocks and their interactions with other software components.**

I. Introduction

A large variety of security technology and individual security solutions exists that can be used by system developers to support security in their systems. Thus, in principle, a developer should be able to take security design decisions early in the development process and just chose from the large set of available security solutions. However, this idealized view of development processes is only valid in very rare cases. The usual approach is a mixture of more-or-less systematic security requirements analysis, ad-hoc design decisions, some best practice, individual security expertise and finally step-by-step improvement after weaknesses have been found either by security testing or in the deployed product.

Another interesting aspect is, that only a rather small set of available security solutions is actually used in real-life products and is obviously not available in the current design processes. One prominent example is the Trusted Platform Module (TPM) as specified by the Trusted Computing Group (TCG) [2]. A very interesting work [3] describes TPMs as the future elements for security. This security chip is already available in thousands of notebooks and laptops. A TPM potentially can provide various security functionalities that can be used to secure network connections, monitor the security of devices, securely store data, etc. However, only a very small subset of installed TPMs is actually activated and used. Furthermore, if it is used at all mostly a very small subset of available functions of the TPM is involved. Experience with developers interested in applying the TPM has shown that one of the reasons for the missing uptake of this technology is the complexity of the specifications. It is not clear how combinations of TPM commands (or TCG software stack commands) can actually implement some particular security services. Thus, making advanced security functionality available for development processes is a challenge.

Another challenge is the correct identification of security properties that a particular solution (e.g. a protocol in a particular configuration) can provide. Developers usually are not security experts. Best practice guides can distinguish insecure configuration of a security protocol from secure configurations. However, they usually do not provide sufficient information on the *security service* the protocol can provide to the application where it should be integrated in.

In a realistic development process, several roles need to be considered. The core role is the actual developer (or group of developers). They need to understand security requirements and they need sufficient information in order to take correct security design decisions. The task of the developers can again be separated into those that create the specifications (or models in a model-based process) and later stages of the development. Security design decisions should already be taken early in the process, but details, configurations etc. also need to take place in the actual development phase. Thus, there also needs to be communication between the two phases. As developers usually are not security experts there are two more roles to be considered. The first can be called the security solution expert. This person creates or validates security solutions and can tell exactly which security properties a solution can provide under which assumptions and constraints. Second, there is the domain security expert. This person has deep knowledge of a security-relevant aspects of a particular application domain. Thus, the domain security experts can identify threats and risks in a domain and is able to decide which security solutions can be applied in the domain and which configurations are suitable.

This paper concentrates on the security solutions expert. In particular, this paper discusses the question how security solutions can be made available to domain security experts and developers. A first pragmatic approach to make security solu-

978-0-7695-4938-5/12 $26.00 © 2012 IEEE
DOI 10.1109/BioMedCom.2012.28

tions accessible and understandable are security patterns. In the context of TPMs, a security pattern can be used to describe on a rather abstract level what kind of security services a TPM can deliver. However, technical details and implementation details necessary for development are usually not included in the concept of security patterns [4], [5]. The approach presented in this paper uses the idea of security patterns and extends it with *Security Building Blocks* (SBBs) represented by UML models in order to describe the functionality and characteristics of security properties in a real world scenario. These SBB models reflect security related software components, which are encapsulated abstractions of program functionalities. Software abstraction, encapsulation and information hiding build the basis of those SBBs. The main focus of using Building Blocks has always been reuseability, maintainability and documentation. An interesting work [6] describes these basic concepts with relation to General Building Blocks used in software development. Consequently, this work tries to refine those general concepts and apply them in the field of security, to model and build more secure systems.

The following section provides a brief introduction to one suitable security engineering process as it was defined by the SecFutur process. Then, a metamodel for SBBs is defined and explained using some simplified examples. In the SecFutur process, SBBs are used to describe various more-or-less complex security solutions, e.g. based on trusted computing technology[1].

II. Security Engineering Process

The development of systems composed of embedded components is a very complex task due to their specific characteristics and nature. Many systems of embedded components are composed of a lot of different embedded components, such as the smart metering system. In these systems, many smart metering devices obtain the information of metering from many houses, process and send it to a different node. This node checks, process, stores, etc. the information and send it to another node, which works with the information provided from many nodes as that one. The systems of embedded components has a reactive nature too. When they process information they may need to react in a specific way, e.g. activating other systems, sending information, etc. One example is the forest control system, where, if they detect a fire, they have to send an alarm. Following this last example we can see that these systems have a real-time nature. They obtain the information, process it and work in real-time. For example, in the mesh ad-hoc network, the nodes enter and exit the system without warning, so the system must react in real-time to these changes and act accordingly. These systems use many components and resources, being hardware or software. For example, they can work with external components such as transmitters, video cameras, sensors or resources such as key-stores, databases, APIs, TPMs, etc.

[1]SBB Models are in general not limited to this particular engineering process.

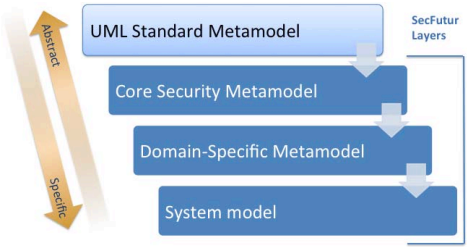

Fig. 1. SecFutur Layers

Companies usually do not follow an engineering process for the development of these systems. Their way of work is start developing as soon as they can. They sometimes follow a methodology but their focus is to start developing and adding functionalities as they find it necessary. This implies that security is either implemented latter in the process as an extra feature or just ignored. Sometimes, when the functionality of the system is almost complete they start adding security. Obviously, in this stage, security is not naturally integrated in the system.

SecFutur proposes a security engineering process that allows to develop and use security solutions in order to satisfy the security requirements of systems of embedded components. It integrates, in a flexible way, security solutions in a framework for the development of systems composed of embedded components. Its main objective is to help developers and engineers in the management of security aspects and its use in System Models. The process can be applied to existent processes, improving the security functionality of any process used to model a scenario. Due to size limitations it is not possible to explain all the details and characteristics of the SecFutur Security Engineering Process. A more complete work can be found at [7].

Some of the most important characteristics of the process are:

- It helps system developers in making design decisions for finding the best solution for their systems
- It facilitates the certification and the national / international regulations of the security artifacts
- It satisfies the specific requirements of the systems
- The implementation solutions are provided by means of SecFutur Patterns (SFPs) and Security Building Blocks (SBBs)

A. Artifacts

The different artifacts of the security engineering process have specific objectives and functionalities. Figure 1 describes the artifact structure.

The SecFutur layers go from more to less abstraction and specification. The upper one, the Core Security Metamodel (CSM), is the most abstract. It is based on the UML Standard Metamodel. This metamodel is used as basis for the definition of the different UML elements used in the creation of the CSM such as classes, relations, attributes, etc. The CSM defines the grammar and language for the definition of the

domain-specific security artifacts. Because of that, the CSM is domain independent and only defines the abstract architecture. The specification of the security properties and characteristics of each domain is done in the Domain Security Metamodel (DSM). This one uses the CSM as basis because it defines the language. Finally, the System Model is the most specific layer. It is the model of the use case. In this model the system engineer imports a DSM (or various DSMs) and apply its security properties in order to fulfill the security requirements of the system. As we said before, due to the size limitations the reader can find more information of these layers in [7].

The DSMs, as we explained before, define the specification of the domain security knowledge. It allows experts to capture their security knowledge (properties, solutions, threats, etc.) related to specific scopes (standards, company policies, etc.) in a specific domain (MANET, Smart meters, etc.). The security properties defined in a DSM are related to implementations by means of SecFutur Patterns (SFP) and Security Building Blocks (SBBs). The security properties define the characteristics and attributes of the solution and the SFP and SBBs their implementation using software / hardware elements. A SecFutur Pattern is a evolved version of the traditional security patterns [4], [5], adapted and extended to the SecFutur Engineering Process. It provides informations such as the security properties provided and the elements of the system where they can be applied, some examples of use (with support for computer-processing), the elements of the system model that must inter-operate with the pattern elements, a list of restrictions and metrics of the pattern with regards to the security properties defined before, the elements that must be added to the system in order for the pattern to work (this part is done by using the Security Building Block Models (SBBMs), a series of rules (in OCL format) used to verify the sound integration of the pattern in the system, a series of assumptions that apply to the system once the pattern has been integrated, some known uses and finally related patterns. Due to size limitations we only do a superficial description of this element. Thus, each security property is attached to a SFP, which defines its implementation by means of a SBBM and SBBs. The Security Building Block Models define the structure, relations and elements of the solution. Its basic elements are the SBBs. A SBB (or the composition of several SBBs) can provide the implementation solution for a specific property in a specific DSM. Following we describe these elements, its characteristics and functionality.

The creation of a DSM involves two different steps. First the analysis of the domain and second the definition of the different security properties. Although the two steps are out of scope of this paper we describe them briefly so the reader can understand better how the Security Building Block Models and the Security Building Blocks provide solutions to a great number of security properties of different domains. Briefly, the analysis of the domain checks the possible security threats and security properties of the system. This analysis provides the necessary information for the definition of the security properties. Once the security domain expert has the information

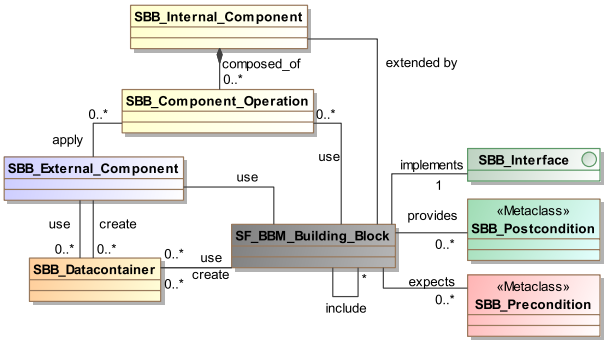

Fig. 2. SBB Metamodel

of the domain, she starts modeling security properties. Each security property is composed of several elements such as its threats, assumptions, certifications, V&V elements, etc. After the security domain expert defines a security property she searches for the SBB model that can provide a solution. The search of the solution is done by checking the characteristics of the security property in the list of SBB models. The SBB models are defined by the security property they fulfill, some requirements of the system (such as the elements they need in order to work correctly), the domain, etc. When the SBB model is found, it is attached to the SFP and then linked to the security property.

III. Security Building Blocks

A. Security Building Blocks Basics

In contrast to a security pattern, one single Security Building Block does not describe a complex integrated security solution. SBBs should be seen as encapsulated components that are domain-independent and can interact with other components in order to provide a clearly defined security service. The concept of abstracting software functionalities in SBBs can use other SBBs and can also interact with other components in a clearly defined way. In principle, a SBB can be just a concrete implementation of a security solution. However, in order to integrate a SBB into the engineering process, a description of the SBB is required. Here, this description is done in terms of a UML model. Thus, a so-called SBB Model represents one (or several) instantiations (i.e. implementations) of the SBB.

As SBBs may reflect concrete implementations of a Security Solution, they also need to provide an interface which defines method names and data types used by the SBB. On one hand this is documentation for the system modeler, to better understand how the SBB can be integrated in the system-model. On the other hand it serves as a concrete specification how the SBB may be implemented in a concrete realization. In addition to the security properties (or security service) provided by the SBB, this model also needs to provide information on preconditions and constraints, as well as on postconditions on the system that need to be fulfilled after the SBB was applied.

The SBB Metamodel, which is described in detail in Section III, defines all the different artifacts and their relationships which were concisely presented in this section.

B. Security Building Block Documentation

As explained, SBBs describe encapsulated software components which are more-or-less security relevant and should be considered in the development process as early as possible. While most developers and engineers are no security experts, they need additional help and information how to actually use the SBBs. For many application fields, where SBBs may be used, additional documentation must be provided to support the development of more secure systems. This is exceptionally important if a SBB encapsulates a software component which is based on communication which involves two or more parties (e.g. cryptographic protocols, hand-shake, key exchange, ...).

For this cases, which occur rather often, other diagrams (Use Case Diagrams, Activity Diagrams, Sequence Diagrams, etc.) may be used to clarify how the SBB may be applied successfully and secure. Additionally documentation may be useful that describes the behavior of a SBB non-formal in simple written text.

IV. The Security Building Block Metamodel

The Security Building Block Metamodel (SBB Metamodel), as shown in Figure 2, delineates artifacts and relationships to construct Security Building Block Models (SBBM). More concretely, the SBB Metamodel acts as a determining factor to depict one or more Security Building Blocks and their interactions with other artifacts to build a SBBM.

Different artifacts enable the SBBM designer to describe their SBBMs in a concrete way (as a detailed view of internal SBBM components), but also leaves the possibility to describe interfaces which may be used from a system model. Although a concrete SBBM and its SBBs may be used in concrete implementations, its main purpose is helping to represent Security Properties at the implementation level. These implementations provide the solutions of the security properties defined in the Domain Security Metamodel. The solutions are specified by means of Security Patterns.

Every SBBM and so their SBBs come with their own dependencies and conditions. These dependencies may be whole system components (databases, key-chains, TPMs or other external components), simple data structures (cryptographic keys, plaintext / encrypted data, etc.) or even other SBBs. In a later stage, all these dependencies should be resolved automatically and imported into the concrete system model after a Security Pattern is selected as a solution for a specific Security Property.

The following presents a more detailed explanation of the artifacts shown in Figure 2.

A. Security Building Blocks

SBBs are the key components of any SBBM. Any artifact from the SBB Metamodel has at least one direct relationship to a SBB.

SBBs are used to represent any kind of security-related system components and encapsulate them in a single artifact which is used in the SBBM. A SBB may be defined broadly in an early stage of the SBBM and refined more detailed as soon as more information is needed or provided. Since a SBB may be composed of other SBBs the level of detail may be increased during the SBBM development process when it is needed or required. On the other hand it is also common that SBBs are used to compose a more complex SBB. This is also independent from the level of detail of any single involved SBB and depends only of the desired level of abstraction of the SBBM.

B. Artifacts and Interactions

1) SBB_Datacontainer: The SBB_Datacontainer artifact is the most general type in the SBB Metamodel. It is used as a container for any kind of data which needs to be processed by a SBB. It may appear in the model as an output value of an external component (*SBB_External_Component creates SBB_Datacontainer*) or of a SBB (*SF_BBM_Building_Block creates SBB_Datacontainer*). Additionally a SBB may use it as a input value (*SF_BBM_Building_Block uses SBB_Datacontainer*).

2) SF_BBM_Building_Block: The SBB follows the description from Section IV-A. It acts as the central component in the SBB Metamodel and has relations to any other artifact.

- *create* and *use SBB_Datacontainer* values.
- *implement* a SBB_Interface to be used by a system modeler or by other SBBs.
- *include* (multiple) other SBBs.
- *expect* a *SBB_Precondition*.
- *provide* a *SBB_Postcondition*.
- *use* a *SBB_External_Component* directly.
- *use* a *SBB_Component_Operation*.
- *extend* a *SBB_Internal_Component* to enhance or modify its purpose.

3) SBB_Precondition: A *SBB_Precondition* is an requirement which specifies under what conditions a SBB may be applied successfully (*SBB expects SBB_Precondition*). A single SBB_Precondition represents exactly one requirement, which may be formal or informal. It is designated that for any different requirement a single artifact instance is used. For example if an input value of a SBB is a cryptographic key the SBB_Precondition may determine that its size must be at least 128Bit. Additionally another SBB_Precondition may determine that a random number for the key generation may only come from a source considered secure (e.g. ”/dev/random” instead of ”/dev/urandom” in Unix Systems).

4) SBB_Postcondition: A *SBB_Postcondition* is a statement that is valid if a SBB is applied successfully (*SBB provides SBB_Postcondition*). A successful application implies that any SBB_Precondition was obeyed. For example a SBB which encrypts confidential data under a given key may assert that the output data may be protected against eavesdropping. If a SBB has multiple assertions which become valid after a successful application, each different statement must appear as a single artifact instance. Again a SBB_Postcondition may be formal or informal.

5) SBB_External_Component: A *SBB_External_Component* is a system component which must be available for a SBB to function properly, but lies out of scope of the current SBB or even the SBBM. A SBB may use the functionality of the external component either by using its functionality directly (*SF_BBM_Building_Block uses SBB_External_Component*) or by using data structures that are produced by it (*SF_BBM_Building_Block uses SBB_Datacontainer created_by SBB_External_Component*) . If a Security Pattern is selected which involves external components the system engineer is informed about what specific external components are needed. Either the system engineer must provide these components from within their own system model or they are created automatically represented by additional interfaces or even concrete implementations. Another possibility is that a SBB_Datacontainer is used by a SBB_External_Component as an input value *(SBB_External_Component uses SBB_Datacontainer*).

Additionally an SBB_External_Component may apply functionalities of an SBB_Internal_Component, by using its SBB_Component_Operations (*SBB_External_Component applies SBB_Component_Operation*). For example using a SBB implementation which involves a TPM, many functionalities are based around the reporting of a system state. To keep track of the system state, an external component, namely IMA (Integrity Measurement Architecture) is used. IMA applies an operation of the TPM, which modifies internal registers in the TPM. These registers, which reflect the current system state, are later used in SBBs which need this system state for their own functionality. In consequence it is mandatory to distinguish between external and internal components. Both components must be available, but an external component represents a part of the system where the SBBM or a concrete SBB has no direct influence.

6) SBB_Internal_Component: The *SBB_Internal_Component* represents a part of the SBBM which is directly associated with a SBB over its SBB_Components_Operations. Any internal component consists of SBB_Component_Operations, which represent a concrete functionality of the component. A SF_BBM_Building_Block may modify a SBB_Internal_Component (*SF_BBM_Building_Block modifies SBB_Internal_Component*), which is considered as an unspecified usage of a SBB_Component_Operation within the SBB.

Additionally a SBB may modify a internal component (*SBB_Internal_Component extended_by SF_BBM_Building_Block*) such that it enhances the functionality of the component. This operation is also not precise and should be explained in detail depending on the enhancement (e.g. with additional diagrams or textual).

7) SBB_Component_Operation: A *SBB_Component_Operation* is an operation of an internal component which may be executed by a SBB (*SF_BBM_Building_Block uses SBB_Component_Operation*). This usage is handled internally in a SBB and is mostly a call of a function or method, provided by a library of the internal component, which executes the components real operation.

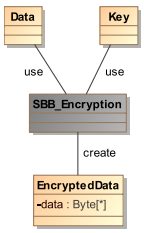

Fig. 3. Simple Encryption SBB Example

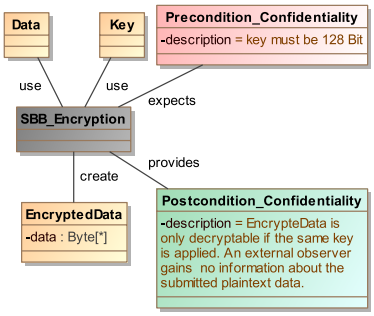

Fig. 4. Added Conditions to the SBB

8) SBB_Interface: The *SBB_Interface* describes the public interface which may be used by a system engineer or other SBBs which need to integrate the current SBB. The application of the SBB is limited to just this specified operations and thus the only way to communicate with the SBB. The SBB_Interface serves as documentation for the input and output values as well as the description of the functionality. This information is mandatory for a system engineer who wants to use SBB in a concrete system model. Additionally the SBB_Interface is used when SBBs are combined with each other. This aspect is described in more detail in Section V-C and V-D.

V. Example Model

Assuming there exists a SBB which simply encrypts data by using a symmetric cipher, as shown in Figure 3. This SBB needs at least data which should be encrypted (Data) and, additionally, a key (Key) that is used to apply the encryption. After the data is encrypted by the SBB, an output value is created which contains the original data encrypted under the given key (EncryptedData). A system engineer, who wants to integrate SBB_Encryption in a system model, needs the information about all input and output values of a SBB. Although the SBB Metamodel defines the general type SBB_Datacontainer, it is necessary that the specific supplied input and output data to any SBB is specified concretely in a SBBM.

A. Preconditions and Postconditions

Describing a SBB only with its input and output values is insufficient in most cases. Therefore two additional artifacts

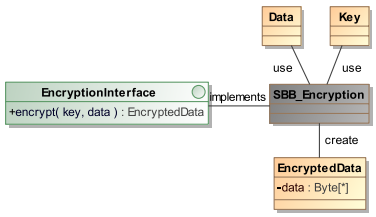

Fig. 5. Interface for Encryption SBB

are used to describe so-called Preconditions and Postconditions. Both conditions are optional and should only be used if there are concrete conditions for the described SBB. Using conditions to describe under which circumstances a software component may be executed and what it provides, is based on the concepts of *Design by Contract*. Two interesting works [8], [9] describe how these conditions can be applied in software design and development.

A Precondition always describes restrictions, which need to be fulfilled, before the SBB may be applied successfully. Thus they represent a requirement information for the system engineer. In this example, as shown in Figure 4, the SBB_Precondition states that the key used for the encryption must be exactly 128 Bit. If the system uses a key that does not meet this precondition the successful application of the SBB is not guaranteed and therefore the postcondition is not provided. In consequence this means that the system engineer is forced to fulfill all the preconditions of any used SBB in order to obtain the postconditions they provide. On the other hand a SBB_Postcondition describes what the SBB provides if applied successfully. In this example the SBB_Postcondition_Confidentiality states that EncryptedData is now encrypted under a specific key and may only be decrypted if the same key is used. Furthermore it states that an external observer is not able to gain any information about the submitted original data.

If a system modeler now uses EncryptedData (e.g. send it to another system component or over a network device), he is assured that no one without the specific key is able to use the submitted data in any way to gain access to its original plaintext content.

B. Interfaces for SBBs

Each SBB consists of an interface which may be used in a system model to apply the functionality of that specific SBB. In this example the input values `data` and `key` are parameters to the `encrypt()` method. This method results in the output data `EncryptedData`. Figure 5 shows the `EncryptionInterface` of the Encryption SBB. More details on interfaces can be found in Section V-D.

C. Combining SBBs

It is also possible that SBBs are combined with other SBBs to enhance and encapsulate functionalities. There are different ways to express such a combination. One approach is shown in Figure 7. In this case the output of the Encryption SBB

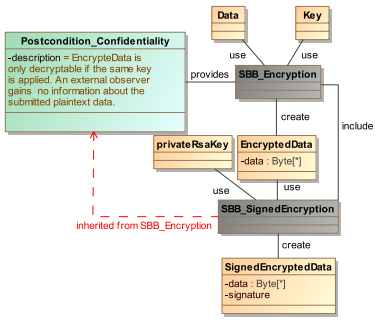

Fig. 6. Combining SBBs Postcondition Inheritance

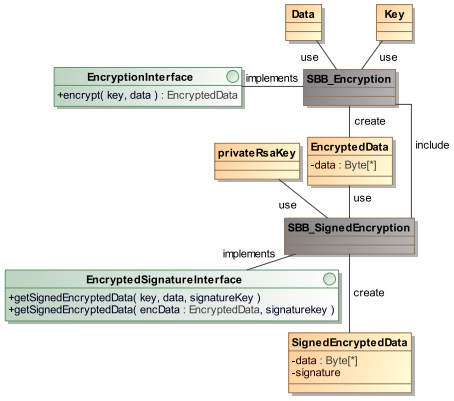

Fig. 7. Combining SBBs

is used in another SBB (SignedEncryption) which generates a signature for that data. The include statements states that the interface for SignedEncryption also accepts the input values from the Encryption SBB. Additionally both SBBs are bound through the `EncryptedData` data-structure.

Another approach to combine SBBs is to use two independent SBBs and include both in an additional SBB as shown in Figure 9.

There exist two SBBs, one (Signature) generates a signature of any arbitrary data and the other one (Encryption) encrypts any arbitrary data. Both may be used independently from another (this is not possible in the previous model from Figure 7, only EncryptedData may be signed there). Both SBBs are included in a third SBB called SignedEncryption. This SBB now uses the SBBs (Signature and Encryption) to generate also an `SignedEncryptedData` data-structure. While in both cases the resulting data-structures are semantically the same, both modeling approaches are different. In the latter case the SignedEncryption SBB is not directly bound to the `EncryptedData` data-structure, which means that `EncryptedData` is no valid input parameter for this SBB by default. (In this example it might be legal to add a separate method which also accepts `EncryptedData` as an input parameter. This decision is left to the SBBM designer and depends on the concrete SBB.)

1) Inheritance of Included Conditions: Another important aspect while combining SBBs is that postconditions of in-

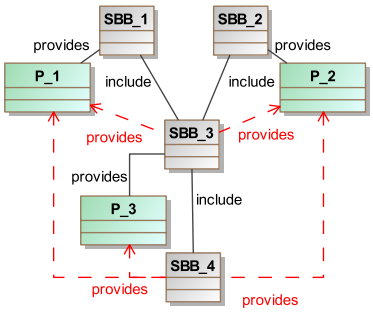

Fig. 8. General Postcondition Inheritance

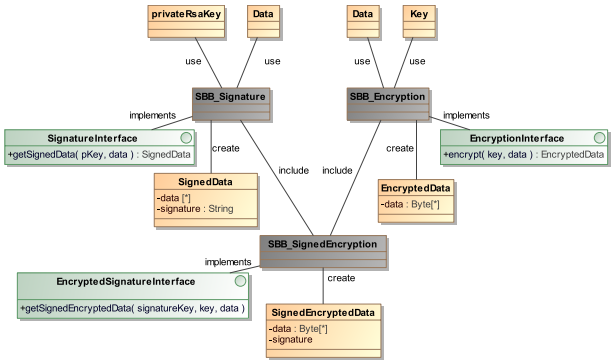

Fig. 9. Combining SBBs only with Include

cluded SBBs are maintained throughout the inclusion chain.

In the example shown in Figure 6 this inheritance means that the output value of the SignedEncryption SBB (`SignedEncryptedData`) maintains the Confidentiality Postcondition from the Encryption SBB. Thus, a postcondition can always be seen as a statement which is valid for an output value of the particular SBB[2].

Figure 8 delineates a more general approach, how the inheritance is realized among deeper and nested hierarchical structures.

Whenever a SBB includes one or more other SBBs (SBB_3 includes SBB_1 & SBB_2 and SBB_4 includes SBB_3) (as a child) the including SBB also provides all postconditions its parents provide. Table I shows all SBBs and their provided postconditions, where an ”o” marks an inherited postcondition and a ”x” marks a directly associated postcondition.

TABLE I
INHERITANCE OF CONDITIONS

	P_1	P_2	P_3
SBB_1	x		
SBB_2		x	
SBB_3	o	o	x
SBB_4	o	o	o

A child SBB may always add additional postconditions, as shown for SBB_3 in Figure 8, but these direct postconditions do not affect the postconditions of their parents.

[2]This is valid even if no output value is shown explicitly in the model.

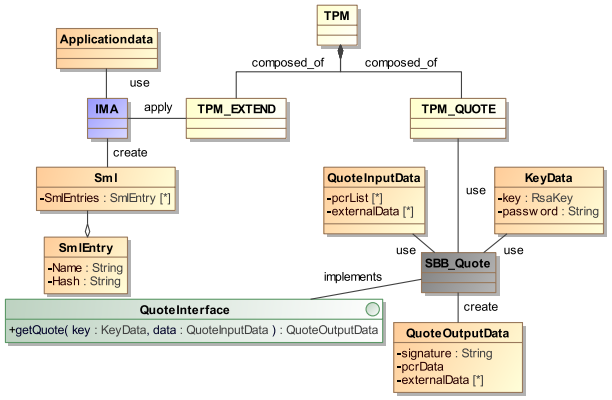

Fig. 10. Internal and External Components

While using the SBBs in a specific system-model, this means that a SBB may be substituted by a SBB which is deeper nested in the same hierarchical inheritance structure (covariant behavior [10]). For example SBB_2 may be substituted by SBB_3, but SBB_3 must not be substituted by SBB_1 or SBB_2.

Moreover, it is also possible to substitute a SBB with another SBB which is not even in that particular hierarchical structure. This means that if two distinct and independent SBBs provide the same postcondition, they can be substituted by one another.

For preconditions the same methodology may be used. As long as a particular hierarchical structure is observed the preconditions of any SBB in that structure stay the same. As soon as a distinct and independent SBB is able to substitute a SBB (by satisfying the original postcondition), the preconditions may be different to any of the preconditions of the substituted SBB. This means that if a SBB provides the same postcondition, but with totally or partially different preconditions, it may be used though.

D. Interface from System Model to SBBM

A crucial part in the modeling of SBBs is the definition of an interface. This interface is used either from a system engineer who integrates a SBB into the system model or used to interconnect SBBs within the SBBM itself. The interface is the only visible part and thus the only way to communicate with the SBB. This means that if a SBB is applied in a system model the system modeler may only interact with the specified methods of the building block. While this is the standard procedure how interfaces are used in general, it is a crucial requirement as a SBB expert has to consider the interfaces when designing SBBs.

While a SBB itself provides a solution for a specific security requirement, there may also exist different SBBs solving the same requirement but with different other components involved. As long as the interface of any different SBB is equal, the SBB is easily exchangeable during the development process.

Furthermore it is assumed that for any operation a SBB

provides, an appropriate name is used which follows general coding guidelines. This is achieved by naming the operations after their functionality (setKey, loadKey, encryptData, decryptData, signData, getQuote, verifySystemStateReport, etc.). Currently there exists no predefined structure for interfaces in the SBB Metamodel, although it may be helpful at a later stage to specify operations any SBB needs to implement (e.g. for test-cases or input validation).

E. Functional Components

The model differentiates functional components into internal and external components. Whenever a component consists of a complex software-system which execution or behavior is not controlled by a SBB, it should be modeled as a SBB_External_Component. On the other hand, if a SBB directly controls a component and uses specific component operations, it is advised that this operation is modeled as a Component_Operation which can be composed to a SBB_Internal_Component. Depending on the SBBM and the level of detail, a SBB_External_Component may become a SBB_Internal_Component as the SBBM evolves or vice versa. A legal approach is to start off with a SBB_External_Component. As soon as a SBB exists which needs to execute a specific operation of a SBB_External_Component the operation should become a Component_Operation and the SBB_External_Component should be converted to a SBB_Internal_Component composed of the Component_Operations.

1) Internal Components and Operations: Figure 10 depicts a SBB_Internal_Component (TPM) which is composed of two SBB_Component_Operations (TPM_EXTEND and TPM_QUOTE). A SBB_Component_Operation is always encapsulated in a SBB applying the operation. This means it is not allowed that a component operation creates data structures directly by itself. In the concrete example, shown in Figure 10, the SBB Quote takes two SBB_Datacontainer (`QuoteInputData` and `KeyData`) and applies the TPM_Quote operation on that data. The result of the operation is encapsulated in the output data structure `QuoteOutputData` and returned as a result of the `getQuote()` method.

SBB_Internal_Components are used whenever the component is composed of SBB_Components_Operations which are used by applying these operations directly within a SBB or by a SBB_External_Component.

2) External Components: To introduce external components, Figure 10 shows a SBB_External_Component which is called IMA [11] (Integrated Measurement Architecture). IMA represents a collection of different components encapsulated in the artifact instance. One part of IMA incorporates with the internal component TPM over its TPM_EXTEND operation. It takes data (Application Data) gathered from executables loaded in the operating system and applies for each of this data the TPM_EXTEND operation. With each call of that operation, the internal state of the TPM is updated, reflecting the current system state. Additionally this data is stored in another SBB_Datacontainer called SML (Stored Measurement Log), which may later be used to verify the trustworthiness of the system. IMA is represented in the system as an SBB_External_Component because its functionality is controlled externally. Whenever a externally executed executable is loaded into the system memory, IMA is automatically called by the operating system.

As said earlier, a SBB_External_Component is used whenever a SBB does not or is not able to directly control the component and mainly operates with its generated data structures.

VI. Conclusion

The SBB Metamodel and the SBB Models as provided in this paper provide one possible approach towards exact specifications of security solutions and their integration into security engineering processes. A validated security solution can be described in a way that preconditions, constraints, dependencies, etc. are exactly expressed and considered in the integration of the SBB into a system. Thus, SBBs can make security solutions better accessible and in addition, due to the clearly shown dependencies, constraints and conditions, using SBB Models to support security design decisions can also result in a higher overall system's security, as errors in dependencies and integration can either be directly prevented by analyzing integrated SBB Models or found in a subsequent validation of the refined system model. The next step in this work will be to integrate the concept of SBBs with the SecFutur Core Security Metamodel, Domain Specific Metamodels and describe domain-specific integrations of SBBs for the realization of more complex security properties.

Acknowledgment

This work is partially supported by the E.U. through FP7 project SECFUTUR (IST-25668).

References

[1] *Design of Secure and Energy-efficient Embedded Systems for Future Internet Applications (SECFUTUR), IST-25668, Seventh Framework Programme*, www.secfutur.eu.
[2] T. C. Group, "Tpm main specification."
[3] S. Pearso, "Trusted Computing Platforms, the next security solution," HP Labs, Tech. Rep., 2002.
[4] e. a. Steel C, *Core Security Patterns*. Pearson Ed. Inc., 2006.
[5] S. M. Roedig U, "Security Engineering with Patterns," in *Pattern Languages of Programs*, 2001.
[6] M. Lenz, H. Schmid, and P. Wolf, "Software Reuse through Building Blocks," *Software, IEEE*, vol. 4, no. 4, pp. 34–42, july 1987.
[7] J. F. Ruiz, R. Harjani, and A. Maña, "A security-focused engineering process for systems of embedded components," in *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systemss*, ser. D4RCES '11. New York, NY, USA: ACM, 2011, pp. 4:1–4:9. [Online]. Available: http://doi.acm.org/10.1145/2349913.2349917
[8] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, oct. 1992.
[9] Meyer, B., *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, 1997.
[10] L. Cardelli, "A Semantics of Multiple Inheritance," *Information and Computation*, vol. 76, pp. 138–164, 1988.
[11] IBM_Research *et al.*, "Integrity Measurement Architecture," Internet, 2012. [Online]. Available: http://researcher.watson.ibm.com/researcher/view_project.php?id=2851