Checking and Verifying Security Requirements With the Security Engineering System Model Core

Hendrik Decke
Volkswagen AG Group Research
Email: hendrik.decke@volkswagen.de

Jean-Pierre Seifert
Technical University of Berlin
Email: jpseifert@sec.t-labs.tu-berlin.de

***Abstract*—As the need for security engineering methodologies for embedded and/or distributed systems rises several different approaches have been proposed. Especially the automotive sector is pursuing the development of new ways to better consider security in the design process. Nevertheless, most of these approaches are custom-tailored for specific use-cases or application domains and are not applicable for other domains. We propose a security requirements engineering process with a generic system model core, which can be customized with application domain specific extensions. This allows the instantiation of application domain adjusted security requirements engineering methodologies without much effort. Additionally, the generalisation of the system model allows the exchange of checking or verification methods with only a small need for adaptation to new application domains. We present our system model core and demonstrate its extensibility on the example of vehicular systems. We then show two methods for formal inspection of the system model. First, we show how the security engineer can be assisted by consistency checking of the system model, then we show how to verify the sum of generated security requirements to ascertain the correctness of the security concept.**

Keywords–Security engineering; requirements engineering; requirements verification; system model core.

I. Introduction

How to design secure distributed and/or embedded systems is a repeatedly discussed question and it is recognized in most industries that requirements engineering is critical to the success of any development project. How to extend requirements engineering into the realm of security has been proposed with a multitude of approaches. Being able to learn from existing methods in security requirements engineering for software systems, most of these approaches focus on particular aspects of the security requirements engineering problem while additionally focusing on a very specific use case or application domain. Given these results, the question why these approaches did not lead to a broader industry adoption was examined by [1] and [2].

For example, the work in [2] presents three interesting factors for missing adoption of methodologies.

1) The business case for employing security best practices is missing.
2) Developers lack security expertise, which is currently required to employ security best practices.
3) The risk of committing to a particular security approach is too high.

Furthermore, [1] gives a broad range of properties and criteria for methodologies to lead to broader industry adoption. Without repeating all of them we saw the feasibility to improve in three major fields.

Notation By describing the security engineering system model core (*SESMC*) as a generic core, we hope to allow the easy exchange and reuse of varying approaches to different parts of security requirements engineering. This reduces the risk of committing to one particular security requirements engineering approach since additional and conceivably needed features can be implemented and added easily. By adding application domain specific extensions we can demonstrate how a security requirements engineering methodology can be instantiated for the vehicular domain.

Tool Support By designing *SESMC* and the corresponding security requirements engineering process to be implementable as a software tool for the security engineer, we improve the ability to master the complexity of a large embedded and/or distributed system, which directly increases the business case for employing a security requirements engineering methodology. Furthermore, the software tool may assist engineers with a lower security expertise.

Formal Methods The use of formal methods and verification leads to increased confidence in the result and can ideally point out outstanding problems in the system model.

A. Contributions

We propose a generic system model core to be used for a security requirements engineering process to create methodologies for different application domains. For this system model core, we present methods to perform consistency checks and to allow the verification of a broad range of properties of the system model, including the new concept of non-traceability, with the use of the ProVerif tool [3]. When appropriate, we will relate our elucidations to possible uses in vehicular systems.

B. Structure

Section 2 presents the background and related work. In Section 3, we present the generic system model core and the security requirements engineering process. Section 4 describes the consistency checking of our system model core. Section 5 presents the steps that lead to the verification of a set of security requirements contained in our system model core. Section 6 presents the conclusions.

II. Related Work

As already mentioned, several examples of security requirements engineering methodologies have been evaluated to create our generic system model core. Likewise, methodologies in the field of access policy engineering and model-based software design have been considered as to discover basic approaches to recurring problems in engineering to lay the groundwork for our approach.

The most influential paper for our work is [4]. They present their **Wor**kbench for Model-based **S**ecurity **E**ngineering (WorSE), which supports the modelling and verification of access control policies. Their main contribution is a domain-independent abstraction for security policy design and verification, which they call the security (model) core. This core may be extended for domain-specific analysis, just as subclasses can be derived from super classes in object-oriented software design. As the formal semantics of this security core are clearly defined any extension can still rely on all properties of the parent core, which yields an interesting way to allow domain-specific analysis in a flexible workbench.

Again in [5], our concept of a generic but extensible system model core is motivated. They argue that in the field of (pattern-driven) security methodologies for distributed systems over a dozen approaches have been presented, but with respect to system applicability these approaches are either highly specific or generic. Likewise, they motivate to position a security methodology in the early development phases (analysis and design), because *”This is where all security countermeasures are planned [6], as well as where, according to [7], approximately half of all major security flaws can be prevented. [8]”*

In [9], [10], [11], a methodology for the security engineering and partitioning of hardware/software systems is presented, which is the most similar methodology to our own. It is named AVATAR, extending on the Object Management Group’s Systems Modeling Language (SysML) [12], and enriched with artefacts for security engineering and implemented in the workbench TTool. The authors position the process in parallel to the hardware software partitioning of the y-chart approach [13] so that the asset, threat and security requirement identification can be done in the early design steps of the systems development life cycle (SDLC).

AVATAR is requirement-driven. Given some security requirements, which are designed in separate diagrams and kept in parallel to the application model of the target of evaluation (TOE), the threats and corresponding attack trees can be subsequently defined. Possible risks are then manually annotated to the security requirements, so that the decision for chosen security mechanisms can be documented. After the system model has been finished, the non-security relevant properties can be model-checked using UPPAAL [14] and the security-requirements (confidentiality and authenticity) can be proven by the ProVerif toolkit [3]. ProVerif [3] is a security protocol analyser. It is able to proof security properties over given cryptographic protocols assuming the Dolev-Yao attacker model [15] with very small resources.

Our own approach, which will be presented in Section III, however is goal-oriented and focuses on keeping all information inside one cohesive system model. With the addition of protection groups and threat hierarchies we hope to enable the design of a whole system inside one model without an unproportional increase in size and complexity. It should be noted that, although we do not present methods to check

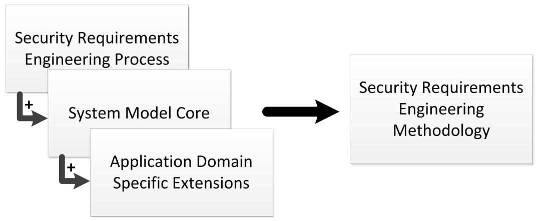
Security Requirements Engineering Process
+
System Model Core
+
Application Domain Specific Extensions
Security Requirements Engineering Methodology

Figure 1. Creating a methodology from the process and system model core.

non-security relevant properties like liveness or computational intensity, we deem our approach to be applicable to these properties and that model-checking, i.e., by using UPPAAL, should be feasible. We increase the amount of usable protection goals in comparison to AVATAR in our version of a ProVerif export. The different possible queries will be presented later in Section V.

III. A GENERIC ONTOLOGY FOR SECURITY REQUIREMENTS ENGINEERING

For our approaches of consistency checking and verifying security concepts to be adaptable to different security requirements engineering methodologies, we define a generic **se**curity **s**ystem **m**odel **c**ore (*SESMC*) with a corresponding ontology. To implement our methods for consistency checking and verification for an existing security engineering methodology the required artefacts have to be identified and matched to *SESMC*. With providing *SESMC* we hope to present a useful mechanism to exchange property checking, property verifying and other algorithms between different security engineering methodologies of all kinds.

Additionally, *SESMC* can be used to instantiate new security requirements engineering methodologies from our proposed process by providing application domain specific extensions. We see our core as adaptable to different application domains, since we only define the recurring artefacts needed in all security engineering domains. By adding the application domain specific characteristics, i.e., communication medium restrictions, cryptography costs or specific attacker models, it is possible to construct an applicable methodology from our core. This is shown in Figure 1. An example of an instantiation for the vehicular domain will be described at the end of this section.

First, we will define the artefacts of our system model core, then we will show how these artefacts are used in our requirements engineering process.

A. System model core

Our system model core (see Figure 3) consists of model artefacts, risk analysis artefacts and security requirements artefacts.

At the centre of the security requirements engineering process are the data instances. They are the assets that should be protected and which are used in the dataflows and processes of the TOE. We allow to define several types of data, while we expect to at least differentiate between data instances representing cryptographic keys (secret and public) and common payload like sensor readings or calculation results. We propose to add a special type for person-related data, like address data and names, so that privacy goals and issues can be easier identified.

1) Model artefacts: The model artefacts consist mainly of executing units, which represent the different nodes, roles or tasks of the system, and communication mediums, which connect the executing units to allow dataflows. Both can be provided with different application domain specific properties (e.g., tamper protection or cryptography acceleration) to allow for more precise analysis. The connections between executing units and communication mediums are consequently named connections and they are used by the dataflows to transfer entities of data. Lifecycle phases represent different stages in the lifetime of a system, which build on one another. They allow to model and analyse a system before or after particular points in its lifetime, e.g., production, operation mode 1, operation mode 2 and decommissioning. Each executing unit has one knowledge set for each lifecycle phase defining the initial knowledge before every process and dataflow of this lifecycle phase.

The dataflows and processes themselves are different for each lifecycle phase to allow modelling varied behaviour. The dataflows are executed concurrently by the executing units but they consist of ordered dataflow steps to structure the communication in time for each dataflow. Each dataflow step describes a message consisting of data instances, which is communicated over connections between two executing units.

To represent the tasks implemented by the executing units we define one or many processes on the executing units, which are likewise executed concurrently. The processes consist of actions, which we will assume to adhere to certain assumptions. The actions describe the processing of data instances inside the executing units. Typical examples for actions are *send, receive, create, save and load*. We allow different notations for the processes and actions (i.e., pi-calculus [16], kripke structures [9] or UML activity diagrams [17]) as long as they satisfy certain properties; e.g., the notation can be translated into a directed acyclic graph and some recurring patterns can be identified. We will discuss the assumptions and properties in more detail in Section V. There is at least one process on a executing unit for each dataflow it is part of to implement the send and receive actions and to synchronize dataflows and process. More processes per executing unit are possible to allow asynchronous processing. To allow for a more precise reflection of reality executing units and communication mediums can be grouped to define hierarchies and boundaries (organisational and/or physical) in the system.

For executing units and model groups we allow to define a multiplicity value to model the replication of nodes, which can be useful to model sets of identical nodes (executing units) or reoccurring patterns of nodes (model groups). The possible values for multiplicity are up to the designer of the methodology to regard the needs of the application domain.

2) Risk analysis artefacts: The model artefacts are annotated with the risk analysis artefacts. The protection goals define which protection goal categories (properties) should be ascertained for which data instance. To be exact we recommend to define distinct protection goals for one data instance in different parts of the system model. We call these parts *Protection Groups* and they cluster data flow steps and actions and allow to propagate the protection goals to all influenced model artefacts (e.g., the sender or executor of a dataflow step or an action). This remedies the need to define confidentiality for each single dataflow step and all executing units of a large dataflow, as all the individual parts of the dataflow can be encapsulated inside one protection group for which one protection goal is set. For each type of protection goal category we define the possibility to define options to further elaborate the meaning of the goal. We see our categorization and options as an example of how protection goals could be formulated, so that we can show the translation to verification queries (see Section V). It is up to the designer of the methodology to define the fitting application domain specific goals.

integrity is the property of protection against modification. It defines which node may create or send a specific data instance. As we outline in Section III-C, we recommend to add an access control matrix to the system model to assign write permissions to nodes per data instance. This can be used for access control checking and to further refine authentication permission. For example, a node may only be allowed to obtain asymmetric keys to create digital signatures, if it is allowed to write the specific data instance.

confidentiality is the property of secrecy. It defines which node may gain knowledge of a data instance. Again, we recommend to add an additional knowledge permission matrix to the system model to allow for easy modelling of knowledge permissions.

availability of a data instance may be important for systems to fulfil their function. Therefore, it should be modelled as a protection goal. This property is difficult to ensure and verify formally but should nevertheless be regarded because denial-of-service attacks are a serious threat.

authenticity is a property with many facets. It may be needed for authorization purposes or to ascertain the origin or integrity of a data instance. We recommend to define separately for the data instance, the sender, the receiver and/or executor if authentication is needed. Additionally, we allow to model the need for non-repudiation and the property of freshness as options, which is needed in many situations.

privacy is the property of confidentiality and self-determination of person-related data. This property implies confidentiality but may also add the requirement to be non-traceable. It is stronger because not only the knowledge of a data instance shall be prevented, but also the plain existence of one particular dataflow or process with one specific data instance shall be indistinguishable from a dataflow or process with another data instance. It is meant to prevent the tracking of users or entities by identifying them by the ciphertexts of their personal data. We differentiate between non-interference like it is described by [18], which decides if the attacker is able to notice if a data instance changes in between sessions and non-traceable, which defines that the attacker is not able to recognize if the same data instance is used in different sessions.

The protection goals each possess one damage potential assessment and they are endangered by different threats. The damage potential assessment quantifies the estimated expected

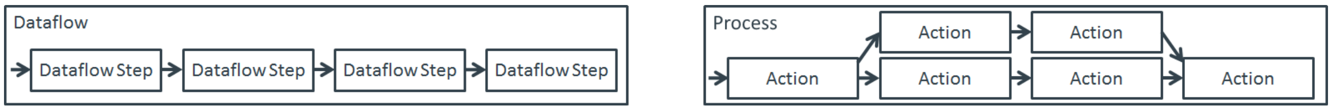

Figure 2. Structure of dataflows and processes

damage when the protection goal is violated by a threat. For each of the threats a risk assessment has to be carried out. We do not dictate any form in which the threats shall be modelled or on how to perform the risk assessment, since the field of threat modelling already offers very elaborate solutions. Merely it must be possible to differentiate between the threats that do not constitute a risk for the protection goals and the threats that do, which will then realise the damage potential.

3) Security requirements artefacts: The security requirements artefacts are then mitigating the threats with relevant risks. We differentiate between security mechanisms and security requirements, which are grouped by a security concept. The security engineer can define multiple security concepts for one system model to allow for easy comparison between competing security solutions. Although most literature subsumes security measures and security requirements under the term *security requirements*, we need to distinguish because of verification precision.

Security mechanisms are direct measures refining dataflows and processes to represent implementations of cryptographic, physical or organisational means to increase security, whereas security requirements are textual requirements that dictate the mitigation of a risk or threat. Only cryptographic security mechanisms can be verified with ProVerif, because organisational or physical mechanisms and textual requirements do not provide the necessary details for the analysis. These types of requirements are checked informally in the consistency check. Textual requirements can be seen as a fallback or placeholder in situations when the designer of the system does not want to provide all the details for a security mechanism, so he can temporarily mitigate a threat to analyse the remainder of the system. Textual requirements can then be replaced by more detailed mechanisms later.

Security mechanisms must describe which changes they apply to the messages of dataflows steps and the actions of processes. For example, a deterministic symmetric encryption must describe how it is applied to a message (the message x becomes $symmEnc(x, key)$) and this encryption action has to be inserted before the send action of the corresponding dataflow step (the action $send(x)$ is extended to $x2 = symmEnc(x, key); send(x2)$).

In addition, security mechanisms can instantiate specific attribute or node requirements for the system model. For example, a cryptographic measure may require a secure key storage in an executing unit, or the existence of a PKI may be needed when using asymmetric cryptography, which has to be modelled by an application domain specific extension. These attribute or node requirements have to be regarded as they lead to (more) complete systems. These requirements can be checked and not fulfilled requirements can be presented to the security engineer in the consistency check.

How these system model artefacts are generated is up to the methodology. In Section IV and V, we assume that the methodology has been carried out and the complete system model can be used for checking and verifying.

B. Security requirements engineering process

Our proposal for a security requirements engineering process consists of the following steps.

1. **Initial architecture** Here, the TOE is designed. First, the executing units, communication mediums and groups are placed and connected to form the topology design of the system. Then, dataflows and processes are added, with their corresponding dataflow steps and actions, to define the communication and process inside the system. Lifecycle phases may be used to define multiple succeeding behaviours.
2. **Protection goal definition** Now the protection groups can be defined for which the security engineer can then define the desired protection goals.
3. **Threat definition** Given the protection goals it is possible to define the threats against these goals. For each goal a damage assessment defines the amount of estimated potential damage if the protection goal is violated.
4. **Risk assessment** For each threat tree (since they may be hierarchically ordered) the security engineer may choose not to assess the risk of all the leafs of the tree(with the most detailed threats). He may choose to assess on a higher abstraction level to increase efficiency. The assessment is then executed with an application domain specific risk system.
5. **Security concept design** Given the risks of the system the security engineer decides which security mechanisms to add. After adding all chosen mechanisms the requirements for the operation of the system can be defined. Node or attribute requirements may be needed to implement security mechanisms. As outlined earlier textual requirements may be used to mitigate threats, if the definition of actual mechanisms is out of scope for the current investigation. However textual requirements decrease the confidence in verification results, as they lack important implementation details.
6. **Verification** When the system model is finished the defined protection goals can be verified. It may be necessary to add further implementation details to decrease inaccuracies, so that the verification model can be built. This includes defining the exact format of messages (payload, order of cryptographic primitives, etc.), the knowledge of the nodes before the communication and process of the current lifecycle phase and the replication details of the processes (how often a node executes a process), if these details have been ignored or not modelled in the preceding steps. It is important to add as many details as possible, as we use the security protocol analyser ProVerif. ProVerif uses the closed-world assumption, which dictates that only facts that have explicitly been modelled as true are true. Everything else is false and non-existent and can not be used to attack the system.

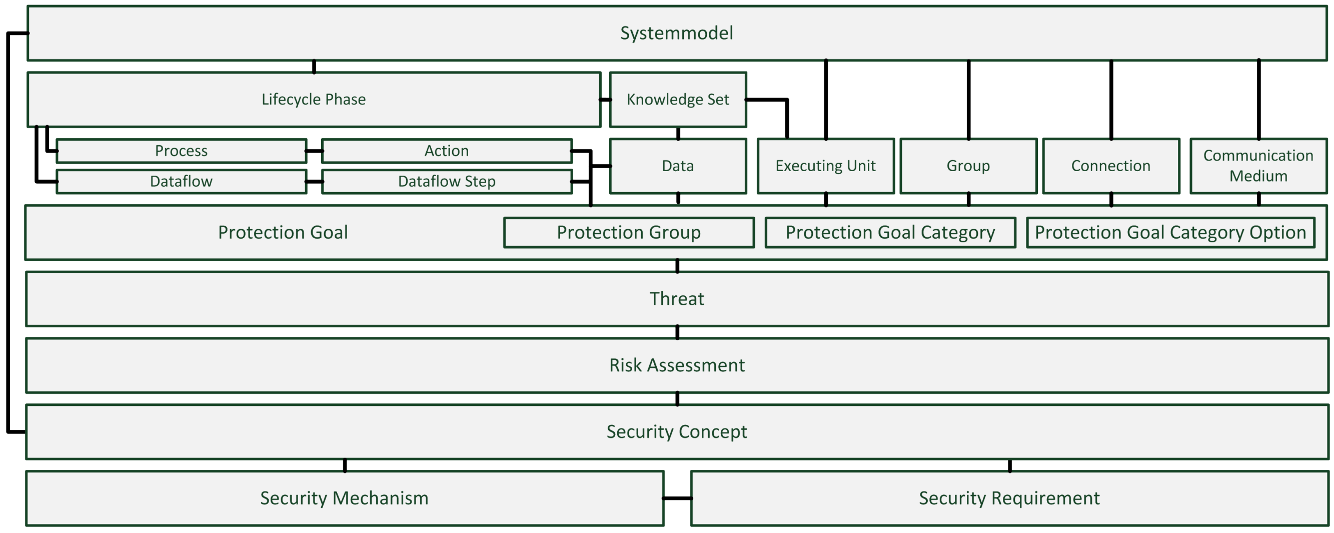

Figure 3. Main artefacts of the system model core.

Additional steps Between the different steps of the process consistency checks may be performed to find contradictions and misconceptions. We describe our approach to consistency checking in Section IV. After the creation of the security concept further analysis and optimization steps may be performed.

Refinement Since several assumptions about the initial architecture of the TOE may be changed throughout the security engineering process the methodology must consider to change parts of the TOE without nullifying all results. Only the artefacts associated with the changes should be marked as invalid and should be analysed again.

C. Matching existing methodologies to the system model core

We formalize our system model core and its extensions with sets, relations and functions, which are representing the artefacts mentioned above. To match an methodology to our system model core these elements have to be identified so that a system model conforming to our definitions can be extracted to use our proposed consistency-checking and verification methods. The following examples illustrates how the system model core is structured and provides an understanding for the following sections.

$\mathbb{E}$ is the set of Executing Units.
$\mathbb{M}$ is the set of Communication Mediums.
$\mathcal{T} \subseteq \mathbb{M} \times (\mathbb{M} \vee \mathbb{E})$ is the relation connecting the executing units and communication mediums, creating the topology graph and representing the connections.
$\mathbb{P}$ is the set of processes.
$\mathbb{A}$ is the set of actions.
$\mathcal{P} \subseteq \mathbb{P} \times \mathbb{A}$ is the relation connecting processes and actions.
$G_{\mathcal{A}} = (\mathbb{A}, \mathcal{A})$ is the directed acyclic graph defining the order of actions (successor relation) with the set of actions $\mathbb{A}$ as nodes and the relation $\mathcal{A}$ as edges.

It can be seen that all artefacts (see Figure 3) are contained in sets. Relationships between these sets are modelled with relations and functions depending on the shape of the relationship. The only exception from the rule is the directed acyclic graph $G_{\mathcal{A}}$, which defines the order of the actions. We omit the remaining sets, relations and functions as they can directly be inferred from the description of the ontology above.

As we outlined in the description of the protection goal categories we additionally recommend to add two access control matrices to the system model to allow for more fine-grained control over write and knowledge permissions. Without the access control matrices the write and knowledge permissions can only be inferred from the dataflows and processes, so that every node that receives or sends a data instance may write or know this data instance. This could be imprecise as some nodes may only need to forward or relay an encrypted message, which could lead to contradictions. Further in the verification steps we can use this matrix to generate more precise secrecy queries.

D. Using the system model core for the vehicular domain

The matching of real world entities in the vehicular domain to the system model artefacts is not as straightforward as one might expect. Especially the application domain specific extensions allow to model an arbitrary amount of detail if desired by the security engineer designing the security engineering methodology. Even the level of detail the security engineer tries to represent in one system model is very dependent on the TOE. Therefore, the extensions have to be chosen very carefully to allow modelling of all possible TOEs and to match the desired level of accuracy in the early design phases. We identified two main kinds of TOEs in the vehicular domain. Either a new control-unit is developed and it is modelled with all its external interfaces or a function is being developed and all involved control-units with the connected external IT systems have to be modelled.

In the first case the different components of the control-unit will be represented by the executing units and all external and internal communication uses appropriate communication mediums. The control-unit itself is represented by a group, which defines the physical border - the plastic shell. In the majority of cases the counterparts at the other ends of the external interfaces (communication mediums) can be represented by a single executing unit. Possible extensions are confined to properties relevant for dimensioning of the hardware capabilities. Examples are the bandwidth or computational capabilities of the communication mediums and executing units and the arising confronting costs by security mechanisms. If standardised communication mediums are used (e.g., IEEE 802.11) several default security mechanisms (read WPA2) should automatically be considered in the security concept as these mechanisms should come at low additional cost. Any special security features of executing units (i.e., tamper protection or hardware acceleration of cryptographic primitives) should also be modelled to enhance the accuracy of the risk assessment.

In the case of a distributed system to implement a new function the abstraction level has to be chosen higher to reduce the system model complexity. It is not feasible to model each control-unit with its internal components. Therefore, in the majority of cases each control-unit is represented by one executing unit. To allow modelling sensors or human-machine interfaces (HMIs) we propose an extension that adds sensors and HMIs as a decorator to executing units. They only need a name and a visual representation in the system model to be regarded in the analysis steps. For example, the modification of a sensor can then be modelled as a threat. These sensors and HMIs can then be referenced in actions and be used to create new instances of data. Hence it is possible to model the information flow of data from the source to the sink. Furthermore, we define a special car group that can only be instantiated once in each system model. It clearly defines the executing units belonging to the car and therefore allows fine-tuning the risk assessment in the later steps of the process as well as defining useful default values for properties like bandwidth capabilities of not yet detailed communication mediums. The executing units outside of this car group may represent IT systems, backends, diagnostic tools, users, other cars (abstracted as a single executing unit) or one of many other possibilities.

In each case we propose a special action to model the impact on external assets. If this action is executed it has an impact on one or many external assets like the quality of the driver assistance systems, the locks or the driving status of the car. It is meant to represent the physical interaction of the system model with its environment. As the system model core focusses only on digital IT systems this extension allows to better represent actuators or mechanics. Consequently the damage and risk assessment can be influenced by the existence of such external asset actions.

IV. Consistency checking a system model

As outlined earlier not only the completed system has to be verified but also individual steps, along the way to completion, should be checked so that contradictions, discrepancies and inconsistencies can be corrected immediately and not later on in the design phase. We will call them consistency issues. These consistency issues are mostly not harmful to the security of the system directly, but they do not allow a precise analysis and verification because the resulting model of the system will not be implementable or at least the needed changes will be more expensive later.

There are several categories of possible consistency issues for which we define rules to find them in the system model. There are topology issues, dataflow and process issues, goal and risk issues, security concept issues and general warnings. Conditions for the instantiation of a consistency issue should be defined, which can be presented to the security engineer. These conditions could be checked regularly after each change to the system model or only when the security engineer wishes to check the design. This is up to the methodology, especially because the possible size of the system model can be very application domain specific and the duration to check the rules can not be estimated beforehand. We recommend to allow the security engineer to ignore or delete consistency issue, as they may be false positives. For example, an unused connection may be unused because the missing dataflow will be modelled later.

1) Topology and process issues: Consistency issues in the topology or the placement of processes are usually remnants of old versions of the model, which were not aligned with changes. They lead to an unconnected system model where single connections and processes are existing without being used or being relevant. Furthermore, the consistency between the dataflows and processes should be checked as for each dataflow step there should be corresponding send and receive actions in the correct order. These dependencies should be automatically resolved when the methodology is implemented as a software tool.

2) Dataflow step and action issues: Here, we aim for inaccuracies and blunders, which may origin from different causes. Perhaps a group of security engineers worked on the same system model and a misconception happened or changes in the system model due to external feedback led to flawed alterations. Examples include the violation of the knowledge and usage access control matrices, if these were implemented, or a data instance originates from multiple sources.

3) Goal and risk issues: These issues try to hint at misconceptions regarding the protection goals and risk assessments of the system model. They draw the attention to different parts of the system model where the comprehension of the goals or risks may potentially be contradictory or even conflicting. This allows to find misconceptions but in addition it allows to check the consistency of different analysis results, that may be produced by different people. In case of a very large system model it may be needed to divide the workload of modelling the security analysis artefacts between several security engineers. After each one has contributed his goals and derived risks these may be checked for consistency to create a consolidated solution. Depending on the application domain different conflicts for protection goals or damage potentials are possible. For example, if confidentiality is required for one dataflow step over a communication medium representing the internet, it should be required for all transmissions over this communication medium. Likewise the damage potential for all these transmissions should be similar.

4) Security concept issues: After the security concept has been built it is helpful to check for the most common blunders. Aside from unfulfilled attribute or node requirements the list of possible issues has to be compiled for each application domain and may include one or many of the following: *sign-then-encrypt vs. encrypt-then-sign, using sensor data as a cryptographic key, missing seed for hash, et cetera.*

5) General warnings: General warnings describe situations, which clearly show potential to be harmful. This includes warnings when a step of the security requirements engineering process has not been conducted yet or unfulfilled node or attribute requirements.

A. Implementing the consistency check

The consistency check can be implemented in many ways depending on the type of the implementation of the methodology. If the process is handled by handwork and manual annotations on paper then a questionnaire could be helpful. If the methodology has been implemented as a software environment/tool the variety of possible implementations becomes apparent. As the structures of the rules are very disparate they do not tend to reveal an obvious object oriented pattern to allow for easy decomposition. We therefore recommend (and chose for our own implementation) to use anonymous functions like lambda functions in C++11 or Java8, which return a string with the result of the rule check or even a more complex object describing possible fixes for the issue, which could be executed automatically.

V. VERIFYING A SECURITY CONCEPT

After the security engineer has finished the security concept for the system at hand he may want to verify if his concept is able to formally ensure the chosen protection goals. We propose a method to transform our system model core to one or many ProVerif [3] models so that the given protection goals can be queried. In Section III, we presented our generic system model core. We proposed that the actions which constitute the processes have to adhere to certain assumptions to allow the extraction of ProVerif models.

1) All actions are nodes of a directed acyclic graph ($G_{\mathcal{A}}$), which as a whole represents all processes. This graph must necessarily be disconnected if there is more than one process. The connection between a sending action and the corresponding receiving action is given by one specific dataflow step, which is recorded outside of $G_{\mathcal{A}}$. The usage of a directed acyclic graph excludes the concept of looping (i.e., while-loops) from our syntax. However this does not exclude to define a single action to represent a while-loop.
2) The creation of a new value for a data instance can be associated with a distinct action.
3) The assignment of a new value to a data instance can be associated with a distinct action.
4) There are distinct send and receive actions or the sending and receiving of data can be associated with other distinct actions.
5) All possible successors of conditional actions (if-else-then, etc.) must be determinable.
6) All actions associated with sending and receiving reference the used communication mediums and data instances or these can be inferred from other sources.
7) For each dataflow step the exact structure of the send message can be inferred. This includes the usage of cryptographic primitives on parts of the message. The easiest solution would be to always use the security mechanisms on whole messages, but more precision is possible.
8) The data instances that are assumed to be known to an executing unit, before a process is executed, have to be modelled. These includes the usage of additional lifecycle phases to distribute the data beforehand or to define the executing unit as the origin of the data instance.

Given these assumptions we can verify the protection goal properties by using the security protocol analyser ProVerif. We divide the verification into four steps show in Figure 4.

1) System model partitioning
2) Process extraction
3) Attacker initialisation and execution
4) Attack trace parsing

For the analysis we allow to model executing units as malign. This means that the attacker may have already gained control over this node and is able to control its behaviour. This can be an important function for many application domains and has consequences for the verification steps. Communication mediums are assumed to be under the control of the attacker according to [15].

A. System model partitioning

In the first step we partition the system model into independent parts. These parts define isolated dataflows and processes, which do not affect any dataflow or process in another part, which can be interpreted as the information flow graph of one specific data instance. This isolation step is important to reduce the size of the verification models. The result of the partitioning step are directed acyclic graphs of actions. The root nodes of one directed acyclic graph are the creation actions for the data instance, or the root actions of processes if the executing unit already knows the data instance and the data instance is used in this particular process.

Our approach takes the directed acyclic graph of one process, which is taken from the relation $\mathcal{A}$ (successor relation), starts with the creation or root action(s) for one arbitrary data instance and then adds additional action nodes, including all the paths that lead to the additional node, when one of the following conditions is met by the path. The next node to check is chosen by arbitrarily selecting one not already checked edge out of the directed acyclic graph of the process under evaluation. The following binding conditions have been defined. Additional processes are checked when the last binding condition is met.

- The action node is a conditional and the condition contains the data instance.
- The action node alters the data instance in any way.
- The action node uses the data instance to create a new data instance.
- The action node uses the data instance for sending or receiving.

In the case of a sending or receiving action an additional edge is added between the two nodes of the directed acyclic

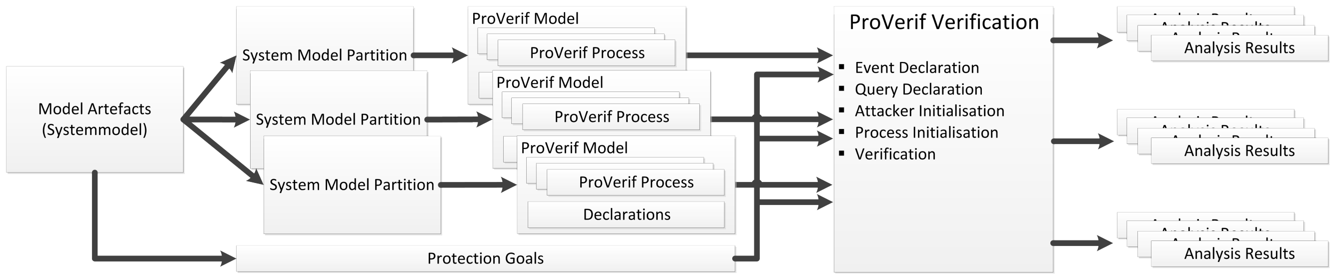

Figure 4. ProVerif Verification

graphs ($G_{\mathcal{A}}$) of the two affected processes. This represents the dataflow step crossing the two processes. The search is then eventually continued in the newly added part of the graph to connect the two processes further. It is important not to discard the information which action belongs to which process (i.e., with using the relation $\mathcal{C}$) as this is needed for the process extraction.

B. *Process extraction*

After the directed acyclic graph (system model partition) has been built, it must be transformed to processes for the ProVerif tool. ProVerif supports several input formats from which we chose the typed pi-calculus as it fits our partitions from the last step. A ProVerif model in the typed pi-calculus consists of a declaration part for channels, functions, reductions, equations, events and queries and the process definitions and instantiations.

The channels can be instantiated directly from the communication mediums with *free channelName: channel.*. ProVerif allows to model a channel as private (by adding *[private]* to the declaration), so that a possible outside attacker can not read any messages of the channel. This property could be controlled by an application domain specific extension that allows communication mediums to be marked as *hidden* from outsiders. It allows to limit the capabilities of the attacker, as he may not be able to control and manipulate all communication mediums simultaneously. If an executing unit marked as malign is connected to a *hidden* communication medium this option has to be removed. The malign node effectively bridges non-hidden and hidden communication mediums, as the attacker may not read from or write to the private medium, but can instruct the malign node to do. Without limiting the generality of our approach we will assume that all channels have been defined as non private, but we will indicate where this option could be relevant.

Next the ProVerif processes for the different processes of the system model partition can be built. Because all processes execute concurrently we will create one ProVerif process for each process. Actions from our system model have to be translated to ProVerif constructs. As we do not dictate a specific notation, we have given assumptions at the beginning of this section, which we take for given for the translation. These leads to a straightforward mapping of ProVerif constructs to the actions.

When these constructs (and possibly others) have been assigned to actions in the chosen notation, the translation to ProVerif subprocesses can begin. First, the actions in the directed acyclic graph can be annotated with the proper ProVerif constructs. Then, the ProVerif constructs are written into the subprocesses regarding the action's membership to processes and regarding the branches defined by conditional actions. At the end the main process is written and all subproccesses are instantiated. The replication of subprocesses (processes) could be an attribute of the system model, which has to be regarded, but without limiting the generality we will always use the replication. How and when events are created will be presented in the explanation of the authenticity protection goal query.

The semantics of cryptography have to be encoded into each ProVerif model, either by adding a library to the command line or by adding all definition to the ProVerif model in the declaration part. Adding them individually to each model allows to only add the cryptographic primitives needed and to allow fine-grained control over the capabilities of the attacker, although in most cases the attacker will be able to execute all primitives in accordance to Kerckhoffs's principle [19] and Shannon's maxim [20]. We recommend to add the definitions individually to decrease the complexity for the solver. The needed primitives can be directly inferred from the used security measures in the system and adhere to the recommendations from the ProVerif manual [18], although we add the session id as an additional parameter to the primitives to model the passing of time and to allow to distinguish between ciphertexts from different sessions. This addition is presented in Figure 5 and further described with the privacy query.

The events mark important steps in the execution of a process and are needed to argue about authentication properties. They are part of the queries, which are needed to verify the protection goals of the system model. For the five presented protection goals four are representable by ProVerif queries including the options we described earlier. Availability can not be proven by ProVerif as it can not be guaranteed by cryptography (alone).

1) Integrity: can not be checked by ProVerif directly, with the meaning defined earlier. The consistency check would already warn if a node sends data, which it has no permission for. All other relevant cases are falling within the authenticity goal.

2) Authenticity: is a very diverse goal. Additionally, to the integrity, which is at the core of this property, it may demand to ascertain the identity of the sender and receiver of a message, to ascertain non-repudiation of sending and receiving, and/or to ascertain freshness of messages. Non-repudiation can be divided into two distinct requirements. It must be ensured that the sender and/or receiver of a message is authenticated and this fact has to be retrievable; i.e., by logging all messages. As such organizational measures can not be proven by ProVerif, we only prove the authentication requirement, but the logging requirement can be modelled by adding an attribute or node requirement to the security concept.

The authenticity property is modelled by events in ProVerif. Depending on the chosen combination of authenticated parties we directly create the events and correspondence assertions as described in [18]. Freshness is modelled by using the injective correspondence. They usually take the form of

query x: key; inj-event(serverTerminates(x)) ==> inj-event(clientAccepts(x)).

This can be interpreted as *if the server terminates while using key x*, then *client accepts key x* has happened exactly one time before.

3) Confidentiality: can directly be translated to a knowledge query: *query attacker(secret).* If knowledge permissions have been defined they can be checked for all nodes without permissions. This can be done by checking if the data instance is send in plaintext over all communication mediums the executing unit under evaluation is connected to. This can be extended to include ciphertexts using keys the executing unit knows.

4) Privacy: is interpreted as non-interference (in accordance with [18]) of the processes regarding the data instance expressed by *noninterf dataInstance.*. As outlined earlier we additionally allow to define non-traceability, which is translated to a knowledge query. If the attacker is not able to find a pair of duplicate ciphertexts (created with deterministic encryption) with the same payload and key from different sessions of the processes, then non-traceability is given.

This can be accomplished with the lines in Figure 5. The types of the data instance and the functions have been chosen for clarity and are not needed to implement the functionality. Type declarations have therefore been left out. We define one function to represent the traceability event and one reduction to allow the creation of the traceability event. When two ciphertexts, which were created with the same payload and key, but at two times, are combined, they create a traceability event. We then let ProVerif prove that if the attacker is able to generate a traceability event, then the two times must be equal. If they are not equal, a violation of the traceability goal was found.

C. Attacker initialisation and execution

After the events and queries have been defined, all that is needed for verification is to initialise the attacker. This consists of building the attacker's knowledge set and capabilities. The knowledge set of the attacker consists of all public names and all the data instances known by the malign nodes. These data instances are sent once over a public channel, before all other processes execute, to allow the attacker to learn them. The capabilities are represented by the functions and reductions known to the attacker. In most cases the attacker will have knowledge of all functions and reductions used in the system. It is conceivable that additional capabilities can be given to the attacker under certain circumstances, i.e., when a cryptographic primitive becomes broken, but this should be a rare case. The ProVerif model can then be analysed and the results from the queries can be retrieved.

```
fun symmEnc(identityType , keyType,
timeType): identityCiphertextType.

reduc forall i: identityType,
k: keyType, t: timeType,
t2: timeType;
symmDec(symmEnc(i,k,t),k,t2) = i.

fun traceabilityEvent(
identityType, timeType, timeType):
traceabilityEventType.

reduc forall i: identityType,
k: keyType, t: timeType,
t2: timeType;
noticeDuplicateCiphertext(
symmEnc(i,k,t), symmEnc(i,k,t2)) =
traceabilityEvent(i, t, t2).

free identityA:
identityType [private].

query i:sid, i2:sid; attacker(
traceabilityEvent(
identityA, new t[!1 = i],
new t[!1=i2])) ==> i=i2.

process (
!(new t : timeType; (
(* instantiate other processes *)
```

Figure 5. Recognizing a tracability violation.

D. Attack trace parsing

Given that the security engineer won't be interested in results without issues, we focus on the found attack traces if they exist. For each property the amount of possible attack traces is rather large, so the parsing has to be flexible. We recommend to condense several attack trace steps to increase legibility and to transport the relevant information. The different lines where the attacker learns all the different parts to start the attack are the first information that can be preprocessed. How to present further parts of an attack is up to the implementer of the methodology.

VI. Conclusion and Future Work

We presented our generic system model core *SESMC* including the corresponding ontology. We believe that *SESMC* allows to create useful security requirement engineering methodologies. Furthermore, it allows to share algorithms and methods on the system model core, which can be extended to use the additional application domain specific extensions. We presented two different applicable methods. The consistency checking of the system model allows the security engineer to check whether the current state has none of the modelled contradictions and misconceptions, which should increase productivity and confidence in the system model. The verification with the security protocol analyser ProVerif allows to formally check the defined protection goals. Although ProVerif is limited by the modelled implementation details it allows to verify the system model at different stages in the design phase up to the first stages of implementation.

We will extend on our work to respect more of the criteria outlined by [1] in the future. To create a complete security requirements engineering process with an accompanying methodology we have to describe the steps of the process and the essential results of each step. In addition, we see possibilities in encapsulation of security knowledge and usage of catalogues to allow for better reuse of engineering results and additional tool-support should be provided for all steps of the security requirements engineering process.

Acknowledgements

The authors would like to thank Bruno Blanchet for answering the questions regarding ProVerif and Jörn Eichler for his useful insights in the state of the art of security engineering.

References

[1] A. V. Uzunov, E. B. Fernandez, and K. Falkner, "Engineering security into distributed systems: A survey of methodologies." J. UCS, vol. 18, no. 20, 2012, pp. 2920–3006.

[2] B. Whyte and J. Harrison, State of Practice in Secure Software: Experts' Views on Best Ways Ahead. IGI Global, 2011, pp. 1–14.

[3] B. Blanchet, V. Cheval, X. Allamigeon, and B. Smyth, "Proverif: Cryptographic protocol verifier in the formal model," 2010, accessed August 30 2015. [Online]. Available: http://prosecco.gforge.inria.fr/personal/bblanche/proverif/

[4] P. Amthor, W. E. Kühnhauser, and A. Pölck, "Worse: A workbench for model-based security engineering," Computers & Security, vol. 42, 2014, pp. 40–55.

[5] A. V. Uzunov, E. B. Fernandez, and K. Falkner, "A comprehensive pattern-driven security methodology for distributed systems," in Software Engineering Conference (ASWEC), 2014 23rd Australian. IEEE, 2014, pp. 142–151.

[6] C. Bidan and V. Issarny, Security benefits from software architecture. Springer, 1997.

[7] A. Jaquith, "The security of applications: Not all are created equal," At Stake Research., 2002, accessed March 24 2011. [Online]. Available: http://www.atstake.com/research

[8] G. Hoglund and G. McGraw, Exploiting software: how to break code. Pearson Education India, 2004.

[9] G. Pedroza, L. Apvrille, and D. Knorreck, "Avatar: A SysML environment for the formal verification of safety and security properties," in New Technologies of Distributed Systems (NOTERE), 2011 11th Annual International Conference on. IEEE, 2011, pp. 1–10.

[10] L. Apvrille and Y. Roudier, "SysML-Sec: A model-driven environment for developing secure embedded systems," Proc. of SARSSI 2013, Mont-de-Marsan, France, 2013.

[11] ——, "Towards the model-driven engineering of secure yet safe embedded systems," arXiv preprint arXiv:1404.1985, 2014.

[12] O. M. Group, "OMG systems modeling language," 2006, accessed August 30 2015. [Online]. Available: http://www.omgsysml.org/

[13] B. Kienhuis, E. F. Deprettere, P. Van Der Wolf, and K. Vissers, "A methodology to design programmable embedded systems," in Embedded processor design challenges. Springer, 2002, pp. 18–37.

[14] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," International Journal on Software Tools for Technology Transfer (STTT), vol. 1, no. 1, 1997, pp. 134–152.

[15] D. Dolev and A. C. Yao, "On the security of public key protocols," Information Theory, IEEE Transactions on, vol. 29, no. 2, 1983, pp. 198–208.

[16] D. Sangiorgi and D. Walker, The pi-calculus: a Theory of Mobile Processes. Cambridge university press, 2003.

[17] S. J. Mellor, M. Balcer, and I. Foreword By-Jacoboson, Executable UML: A foundation for model-driven architectures. Addison-Wesley Longman Publishing Co., Inc., 2002.

[18] B. Blanchet and B. Smyth, "Proverif 1.89: Automatic cryptographic protocol verifier, user manual and tutorial," 2014, accessed August 30 2015. [Online]. Available: http://prosecco.gforge.inria.fr/personal/bblanche/proverif/

[19] A. Kerckhoffs, La cryptographie militaire. University Microfilms, 1978.

[20] C. E. Shannon, "Communication theory of secrecy systems," Bell system technical journal, vol. 28, no. 4, 1949, pp. 656–715.