SysML Models and Model Transformation for Security

Florian Lugou[1], Letitia W. Li[1], Ludovic Apvrille[1] and Rabéa Ameur-Boulifa[1]
[1]*LTCI, CNRS, Télécom ParisTech, Université Paris-Saclay, 75013, Paris, France*
{florian.lugou, letitia.li, ludovic.apvrille, rabea.ameur-boulifa}@telecom-paristech.fr

Keywords: SysML-Sec, Security, Model-driven engineering, Model transformation, ProVerif, TTool

Abstract: The security flaws of embedded systems have become very valuable targets for cyber criminals. SysML-Sec has been introduced to target the security of these systems during their development stages. However, assessing resistance to attacks during these stages requires efficiently capturing the system's behavior and formally proving security properties from those behaviors. This paper thus proposes (i) novel SysML block and state machine diagrams enhanced to better capture security features, and (ii) a model-to-Proverif transformation. ProVerif is a toolkit first released for the formal analysis of security protocol, but it can be used more generally to assess confidentiality and authenticity properties. This paper demonstrates the soundness of our approach using a complex asymmetric key distribution protocol.

1 INTRODUCTION

The growing proportion of connected embedded systems and cyber-physical systems offer more opportunities for attack to cyber criminals. To cite only a few examples of past attacks on such connected systems, we can mention ADSL routers (Assolini, 2012), mobile&smart phones (Maslennikov, 2010), avionic or automotive systems (Hoppe et al., 2011), and smart objects e.g. the recent vulnerability disclosed on the Fitbit (Apvrille, 2015). Security flaws have even been disclosed on medical appliances, such as on the Hospira Symbiq drug pump (ICS-CERT, 2015). Such attacks also target industrial systems whose sensors are increasingly commonly connected with vulnerable information systems, as demonstrated by the Stuxnet, Flame, and Duqu (Maynor, 2006) attacks. The dependability of such systems are targeted with varying objectives, e.g., terrorist acts and ransomware.

System complexity in terms of code size, distribution, and heterogeneity among others is a major risk factor. One solution to better consider the risk of these systems is to take all of their constraints into account, including security, during their development cycle. We previously introduced the SysML-Sec environment to handle the design of such complex systems, in terms of safety, performance, and security (Apvrille and Roudier, 2015). SysML-Sec addresses system development starting from requirements and possible attacks, taking into account software/hardware partitioning. After the partitioning stage, SysML-Sec also supports the design of software components, again with safety and security in mind. TTool is the free/open-source support tool of SysML-Sec (Apvrille, 2003).

TTool relies on UPPAAL for safety proofs, and on ProVerif to perform security proofs from SysML-Sec block diagrams. However, SysML-Sec has several strong limitations both in terms of modeling (i.e., modeling security features), and in terms of proofs (limitations in the models-to-proverif transformations). The paper addresses several of these limitations, with new modeling and transformation approaches. In particular, we have fully formalized and implemented the model transformation.

The next section gives the related work in terms of modeling and proof environment for security. Section 3 presents SysML-Sec. Section 4 focuses on the modeling extensions as well as on the models-to-proverif transformations. Section 5 presents the updated version of TTool and the validation features. Lastly, section 6 concludes the paper.

2 RELATED WORK

Assessing security properties when designing software components mostly relies on formal approaches. For example, (Toussaint, 1993) proposes verifying cryptographic protocols with a probabilistic analysis approach. Protocols are represented as trees whose nodes capture knowledge while edges are assigned transition probabilities. Although these trees could include malicious agents in order to model attacks and threats, security properties are nonetheless not explicitly represented. Moreover, for threat analysis, attacks should be explicitly expressed and manually solved. (Trcek and Blazic, 1995) defines a formal basic set of security services for accomplishing security goals. In this approach, security property analysis strongly relies on the designer's experience. Moreover, threat assessment is not easily feasible.

In more recent efforts, temporal logic languages have been used for expressing security properties. (Drouineaud et al., 2004) introduces a first order Linear Temporal Logic (LTL) into the Isabelle/HOL theorem prover, thus making it possible to model both a system and its security properties, but unfortunately leading to non-easily reusable specific models. (Maña and Pujol, 2008) mixes formal and informal security properties, but the overall verification process is not completely automated, again requiring specific skills. Analogously, the Software Architecture Modeling (SAM) framework (Ali et al., 2009) aims to bridge the gap between informal security requirements and their formal representation and verification. SAM uses formal and informal security techniques to accomplish defined goals and mitigate flaws. Thus, liveness and deadlock-freedom properties can be verified on LTL models relying on the Symbolic Model Verifier (SMV). Even if SAM relies on a well established toolkit - SMV - and considers a threat model, the "security properties to proof" process is not yet automated.

UMLsec (Jürjens, 2007) is a modeling framework aimed at defining security properties of software components and of their composition within a UML framework. It also features a rather complete framework addressing various stages of model-driven secure software engineering from the specification of security requirements to tests, including logic-based formal verification regarding the composition of software components.

More recently, (Shen et al., 2014) developed an expanded UML model extending the sequence diagrams of UML for security protocol verification. Their approach also included translating the model into ProVerif for verification of confidentiality and correspondance. However, our work includes state diagrams for the capability to model a broader range of protocol. Basic sequence diagrams may model only a single execution, while state diagrams may model protocol containing *if* statements and loops. Furthermore, our process includes verification of weak and strong authenticity.

With regard to our previous publications, we propose a way to better model situations (e.g., loops) and their models-to-proverif transformation, taking into account the capabilities and limitations of ProVerif. We thus manage to limit cases where the proof of security properties would fail, without impacting the safety proof capabilities of SysML-Sec diagrams.

3 CONTEXT

3.1 SysML-Sec

SysML-Sec connects goal-oriented descriptions of security requirements and attacks (left section of Figure 1), and the fine-grained representation of assets based on the software / hardware partitioning (right upper section of Figure 1). SysML-Sec also supports phases of the V-cycle after the partitioning stage (lower right of Figure 1), and notably the design of the software-partitioned functions. The design stage includes the definition of security mechanisms, and the refinement of security requirements in security properties to be proved in the design: we address this stage in the paper. Verification takes place at different engineering phases. Simulation is mostly used at the partitioning stage in order to evaluate the impact of security mechanisms in terms of performance. Formal verification intends to prove the resilience of the system under design to threats. Testing serves a similar purpose, but on the deployed implementation resulting from the design models (Apvrille and Roudier, 2015).

A SysML-Sec design is composed of SysML block and state machine diagrams. SysML-Sec blocks can define a set of methods corresponding to cryptographic algorithms, such as $encrypt()$, to be able to describe security mechanisms built upon these algorithms, e.g., cryptographic protocols. Blocks can also pre-share values, a feature commonly needed to set up cryptographic protocols.

The formal proof of SysML-Sec designs relies on UPPAAL (Bengtsson and Yi., 2004) and ProVerif (Blanchet, 2009) respectively for the proof of safety and security properties. ProVerif is a toolkit that relies on Horn clauses resolution for the automated analysis of security properties over cryptographic protocols, under the Dolev-Yao model. ProVerif takes in input as a set of Horn Clauses, or a specification in pi-calculus together with a set of queries. ProVerif then outputs whether each query is satisfied or not. In the latter case, ProVerif tries to identify a trace explaining how it came to the conclusion that a query is not satisfied.

Safety proofs take into account all design elements besides ones specifically defined for secu-

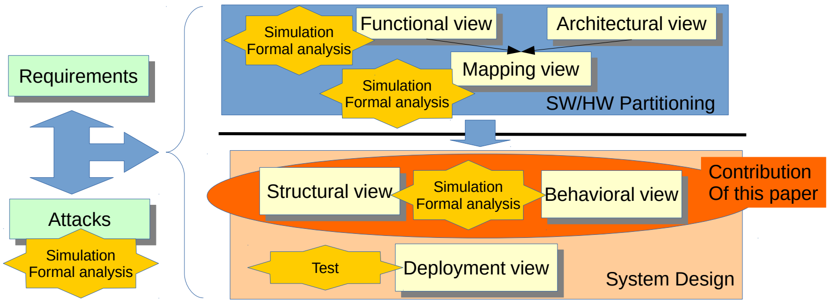


Figure 1: Overall SysML-Sec Methodology

rity purposes: security-oriented pragmas and cryptographic methods that have no impact on safety properties (liveness, reachability). Similarly, the proof of security properties abstracts away irrelevant or non supported system details. For example, the security proof does not require temporal information, so temporal operators (*after* clause) are not taken into account. Other modeling elements, like loops, are ignored by the translation process.

3.2 Missing features in SysML-Sec design diagrams

In the years since we first proposed a translation from a SysML-Sec model to ProVerif, we gathered valuable feedback from devoted testers. Among easily corrected corner cases or usability bugs, some robustness issues indicated more fundamental problems. These issues were often related to a mismatch between the semantics of features in SysML-Sec and their counterparts in ProVerif. We present some of the most important issues that motivated us to completely rework the translation process:

Loops. Often, users take advantage of the verification feature embedded in TTool in order to assess the security of a system that provides a service. For some systems, modeling their state machines using loops seems natural. Unfortunately, ProVerif tries to flatten the given specification as a first step. Using a straightforward translation as it was implemented, verification on a design containing loops would not terminate. We address this significant drawback in this paper.

Private Channels. As in ProVerif, SysML-Sec enables the user to model two types of channels: private and public. While their semantics should be the same for ProVerif, their behaviors with respect to reachability queries differ. ProVerif considers that a message sent should always be received. On private channels, only explicitly specified entities can read the sent message, while on public channels, the message can also be read by the attacker. When assessing the reachability of an event, ProVerif tries to reconstruct a trace and stop at the exact moment when the event is triggered. Therefore, if the event occurs just after a message was sent on a private channel, the message would not yet be read. ProVerif would thus consider the trace invalid, returning a *cannot be proved* result. As we wished, in our context, to provide the ability to prove reachability properties, modifying the private channel representation was also an important issue we addressed in our work.

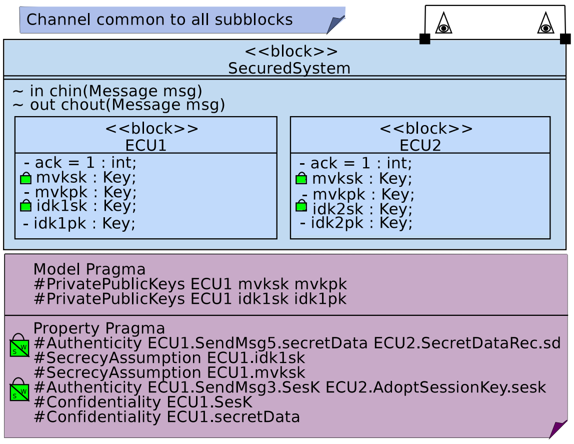
4 EXTENDED DIAGRAMS

4.1 Design diagrams in depth

In TTool, the user first builds the block diagram and associated state diagrams as a graphical model. The block diagram describes the architecture and components of the system, and state diagrams associated with blocks describe their behavior.

We present an example from the European FP7 EVITA project, which defines security architecture for automotive communication systems (Kelling et al., 2009). In this architecture, safety critical ECUs (Electronic Control Units) are interconnected with CAN or Flexray buses. Automotive systems are likely to be attacked either for economic reasons (activating optional features for free), or for criminal purposes. The interconnection of automotive systems to information systems (roads signs, tolls, etc.) and Internet offers new ways to conduct attacks on those systems.

To avoid attacks and their propagation on the ECUs, session keys are regularly distributed among groups of related ECUs. Figure 2 presents the block diagram of an asymmetric key distribution protocol with entities ECU1 and ECU2. An extract of their state diagrams describing their communication protocol is shown in Figure 3. ECU1 processes the key and sends it on a public channel, and ECU2 recieves the message, verifies the key, and uses it to send messages. The locks show verification results to be de-



Channel common to all subblocks
<<block>>
SecuredSystem
~ in chin(Message msg)
~ out chout(Message msg)
<<block>>
ECU1
- ack = 1 : int;
mvksk : Key;
- mvkpk : Key;
idk1sk : Key;
- idk1pk : Key;
<<block>>
ECU2
- ack = 1 : int;
mvksk : Key;
- mvkpk : Key;
idk2sk : Key;
- idk2pk : Key;
Model Pragma
#PrivatePublicKeys ECU1 mvksk mvkpk
#PrivatePublicKeys ECU1 idk1sk idk1pk
Property Pragma
#Authenticity ECU1.SendMsg5.secretData ECU2.SecretDataRec.sd
#SecrecyAssumption ECU1.idk1sk
#SecrecyAssumption ECU1.mvksk
#Authenticity ECU1.SendMsg3.SesK ECU2.AdoptSessionKey.sesk
#Confidentiality ECU1.SesK
#Confidentiality ECU1.secretData

Figure 2: Block Diagram for Asymmetric Key protocol

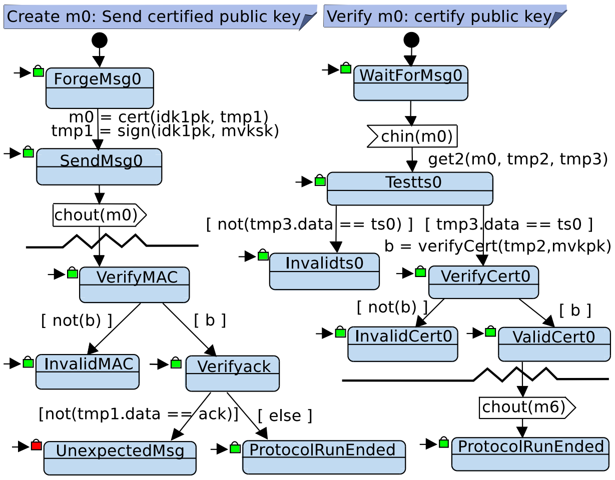


Figure 3: State Diagrams for ECU1 and ECU2

scribed in section 5.

The graphical model is translated for verification in ProVerif, UPPAAL, etc. A detailed formal description is provided in subsection 4.1.2.

4.1.1 Extending SysML-Sec with pragmas

Not all model parameters and queries are stated graphically within the block and state diagram. For example, we needed the capability to declare two attributes equal at the start of a session. We extended SysML with pragmas, i.e. formal text notes regarding the attributes of the system. Pragmas are classified into *Model Pragma* or *Property Pragma*. Model Pragmas provide more security-oriented semantics to attributes of blocks, while Property Pragmas describe security properties of block attributes or state machines. The pragmas are described in Table 1.

4.1.2 Formalization of SysML-Sec diagrams

In an attempt to provide a reliable mathematical base for the proofs performed by ProVerif on a SysML-Sec design diagram, we decided to formally express the transformations performed on the model as it is described in TTool. This formalization will provide understanding on both how we managed to link a general purpose SysML diagram to a ProVerif specification that relies on Horn clauses, and where sources of incompleteness may come from. This formalization is also a first step in building a mathematical proof of equivalence for a future work.

A first transformation translates timer-related capabilities into a set of blocks and signals. The latter are inherently taken into account in our formalization.

Formally, let us denote $\mathcal{D}$ the set of SysML-Sec diagrams; $d \in \mathcal{D} ::= (\mathcal{B}, \mathcal{M}_p, \mathcal{P}_p)$, where $\mathcal{B}$ is a set of *blocks*, $\mathcal{M}_p$ is a set of *model pragmas* and $\mathcal{P}_p$ is a set of *property pragmas*.

We define a block $b \in \mathcal{B} ::= (\mathcal{A}, \mathcal{M}, \Sigma, ss, \mathcal{E}, \mathcal{S}, O, \mathcal{T}, \mathbb{L})$, where $\mathcal{A}$ is a set of *attributes*, $\mathcal{M}$ a set of *methods*, Σ a set of *signals*, ss a *start state*, $\mathcal{E}$ a set of *end states*, $\mathcal{S}$ a set of *states*, O a set of *operations*, $\mathcal{T}$ a set of *transitions* and $\mathbb{L}$ a labeling function that associates attributes, methods, signals and states to their qualified name. Attributes, methods and signals can be mapped to their matching graphical elements in the block view, while states, operations and transitions correspond to the state machine of the considered block.

To each attribute $a \in \mathcal{A}$ is associated a type: a set of concrete values that the attribute may take. We call valuation a mapping that returns a value for each attribute $a \in \mathcal{A}$. We suppose that we are able to construct *patterns* over $\mathcal{A}$. A pattern denoted p is composed of applications of methods or simple operators ($=$, *and*, $+$, etc.) to elements of $\mathcal{A}$.The set of patterns over $\mathcal{A}$ is denoted $\mathcal{P}$. We slightly extend the labeling function $\mathbb{L}$ to accept well-formed patterns as input.

The operations set O is defined to be a disjoint union of sets of elementary operations:

$$O = O_{rdm} \uplus O_{out} \uplus O_{in}$$

where O_{rdm} is a set of *randoms* operations and O_{out} and O_{in} a set of *out* and *in* actions respectively. We encode by $\nu.a$ the *randoms* operations, by $\bar{m}\langle p\rangle$ the *out actions* and by $m\langle a\rangle$ the *in actions* with $m \in \Sigma$, $p \in \mathcal{P}$ and $a \in \mathcal{A}$.

Finally, we define a transition $t \in \mathcal{T}$ as a tuple $(i, o, g, \mathbb{R})$ where:

- $i \in \{ss\} \cup \mathcal{S} \cup O$ is the incoming node.
- $o \in \mathcal{E} \cup \mathcal{S} \cup O$ is the outgoing node.

Pragma	Description
Model Pragma	
PrivatePublicKeys	Two attributes of a block are set as Private Key and Public Key, respectively
InitialSessionKnowledge	Listed attributes have the same value at the start of a session
InitialSystemKnowledge	Listed attributes have the same value when the system starts
SecrecyAssumption	Listed attributes are assumed secret. ProVerif verifies this afterwards
Constant	Declares a string as a possible constant value
Property Pragma	
Confidentiality/Secret	Query the confidentiality of attributes listed
Authenticity	Query the weak and strong authenticity of the two attributes at given states

Table 1: Pragma Descriptions

- $g \in \mathcal{G}$ is a guard, which is a pattern whose valuation is a boolean value. An empty guard is denoted $true$: a pattern that is always evaluated to true.
- $\mathbb{R} = (\mathcal{A} \times \mathcal{P})^{i \in \mathbb{N}}$ is a finite family of actions (pairs attribute and pattern), indexed over $\mathbb{N}$. Each transition can indeed be labeled with a set of actions corresponding to assignments of attributes.

We only deal with well formed blocks so as to map with the intuitive semantic of SysML diagrams. For instance, no transition can reach the start state (*ss*) or come from an end state, and multiple transitions can only join or split at a state and not at an operation.

As we wish to use the ProVerif backend to perform a security proof on a SysML-Sec diagram, such a diagram must first be translated in order to match an equivalent representation that ProVerif can handle.

4.2 Semantics of ProVerif

ProVerif works with two different semantics: Horn clauses and pi-calculus. When translating from SysML-Sec to ProVerif, we generate a pi-calculus specification. The generation takes advantage of the Horn clauses semantic to model concepts, such as loops in the state machine diagram, that are not directly supported by ProVerif.

To enable a more thorough understanding of the proof provided by TTool, one should first familiarize oneself with the ProVerif proof engine. We thus try to briefly present ProVerif in this section and refer interested readers to the manuals and white papers for a better understanding.
Pi-calculus enables description of protocols in term of processes executing in parallel and exchanging messages over channels. Processes can split to create concurrently executing processes, and replicate to model multiple executions (called sessions) of a given protocol. Cryptographic primitives, such as symmetric and asymmetric encryption or hash, can be modeled through functions that are either constructors, which create new values, or destructors, which reduce the number of applied constructors in an expression .

Starting from this specification, reachability, correspondence and confidentiality properties can be queried and ProVerif will present a result to the user that is either *true* if the property is verified, *false* if a trace that falsifies the property has been found, or *cannot be proved* if ProVerif failed in asserting or refuting the queried property. Such failures in proving properties over an unbounded number of sessions and unbounded message space are unfortunately unavoidable since this problem is undecidable (Durgin et al., 2004). Thus, sound approximations are made by ProVerif when translating the pi-calculus specification into Horn clauses.

4.3 SysML-Sec to ProVerif translation

Some components of the SysML-Sec diagram can be mapped to their ProVerif counterparts quite straightforwardly since we borrow the attacker model and the channel semantic from ProVerif. However, for other components, like loops on the state machine diagram or private channels, we had to carefully consider ProVerif reasoning in order to avoid as much as possible cases where the proof would fail. In the following sections, we will present the entire translation process and provide more details about non-obvious features or situations.

4.3.1 Partitions of a SysML-Sec diagram

Since a SysML-Sec state machine diagram may contain loops, we need either to unroll these loops up to a certain limit, or to reuse the part of the translated specification that could be executed multiple times. Both of these choices have been considered in static analysis works and are strategically important, since allowing jumps in the specification often incurs a loss in terms of completeness, while flattening the graph by limiting the number of repetitions affects the correctness of the proof.

In order to maintain the soundness guaranteed by ProVerif, we chose not to limit the loops. As such, the first step in translating a SysML-Sec Diagram composed of well-formed blocks to a ProVerif specification is to determine the initial state of every basic block in every block. A basic block (s,E,S,O,T) is a sub-part of a block that can be considered atomic with only one entry point. From a block $b = (\mathcal{A},\mathcal{M},\Sigma,ss,\mathcal{E},\mathcal{S},O,\mathcal{T},\mathbb{L}) \in \mathcal{B}$ one can construct a set of basic blocks (denoted by $BB(b)$) which forms a partition of the states, operations and transitions sets.

We define $Init_{BB}(b)$ as the set of initial states of basic blocks. This set is a collection of all states that have at least two different incoming transitions.

$$\begin{aligned} Init_{BB}(b) = \{ss\} \cup \{s | s \in \mathcal{S} \wedge \exists (i,s,g,\mathbb{R}) \in \mathcal{T}, \\ \exists (i',s,g',\mathbb{R}') \in \mathcal{T}.(i,s,g,\mathbb{R}) \neq (i',s,g',\mathbb{R}')\} \end{aligned}$$

We give the construction function of basic blocks from traversal of a SysML-Sec Diagram. This function denoted by $\rightarrow$ is specified in terms of a set of inference rules. The construction rules of each basic block begins from its initial state.

$$\frac{s \in Init_{BB}(b)}{(s,\emptyset,\emptyset,\emptyset,\emptyset)}$$

$$\frac{i \in \{s\} \cup S \cup O \quad (i,o,g,\mathbb{R}) \in \mathcal{T} \setminus T}{(s,E,S,O,T) \rightarrow (s,E,S,O,T \cup \{(i,o,g,\mathbb{R})\})}$$

$$\frac{n \in \mathcal{E} \setminus E \quad \exists (i,n,g,\mathbb{R}) \in \mathcal{T}}{(s,E,S,O,T) \rightarrow (s,E \cup \{n\},S,O,T)}$$

$$\frac{n \in \mathcal{S} \setminus S \quad n \notin Init_{BB}(b) \quad \exists (i,n,g,\mathbb{R}) \in \mathcal{T}}{(s,E,S,O,T) \rightarrow (s,E,S \cup \{n\},O,T)}$$

$$\frac{n \in O \setminus O \quad \exists (i,n,g,\mathbb{R}) \in \mathcal{T}}{(s,E,S,O,T) \rightarrow (s,E,S,O \cup \{n\},T)}$$

A basic block is built by adding recursively all transitions and nodes – which are not initial states – that are in a path starting from an initial state.

4.3.2 Translating basic blocks

The translation function of each basic block $(s,E,S,O,T) \in BB(b)$ is given by the function $[\![.,.]\!]$: *state*, *set of transitions* $\rightarrow$ *pi-calculus code*, unfolding the set of transitions into a ProVerif specification. We build the code by concatenating the generated instructions. For this we use the concatenation operator $\oplus$. The translation rules are defined formally in Table 2; they are listed in the order they should be applied. Let us briefly explain the translation rules from Table 2. Rule (a) deals with multiple outgoing transitions, it should be applied when there are more than one outgoing transitions ($|I| > 1$). (b) translates guards and (c) translates actions. (d) and (e) deal with states and (f), (g) and (h) with operations. Finally, (i) stops the translation algorithm. Note that (e) also stops the translation since it deals with transition linking this basic block to an other (more details are given in Section 4.3.3).

When a state is followed by multiple outgoing transitions, it can use any of them as long as the boolean condition that guards the transition is verified. From a security point of view, any trace of attacks using one of these transitions is valid. This is conceptually equivalent to letting the attacker choose which transition to take in case of nondeterministic transitions. We model this in rule (a) by creating values corresponding to each transition, making them public, and waiting for a value corresponding to the attacker's choice.

4.3.3 Pragmas and basic block linking

Rule (e) shows how control is passed from one basic block to the next. ProVerif supports simple process calls by flattening them, and as a result, does not terminate for diagrams containing loops. Thus, instead of directly calling basic blocks, we generate tokens of the form $tok(call\_\mathbb{L}(n'),arg)$ and make them available to the attacker. One such token allows the attacker to execute the basic block whose initial state is $n' \in Init_{BB}(b)$, with the values of the attributes of the block being passed as argument ($arg = \bigcup_{a \in \mathcal{A}} \{\mathbb{L}(a)\}$).

This token should only be used once, and only in the right session, so we add to the token a session identifier to guarantee that the token is forged and used in the same session, and use a nonce mechanism to prevent its reuse. Unfortunately, due to ProVerif approximations – which are unavoidable – using nonce affects the ability of ProVerif to prove properties (mostly strong authenticity) in the presence of loops.

5 VALIDATION IN TTOOL

ProVerif results are passed back to TTool as a text file containing results. Reachable state queries are performed for all states, while authenticity and confidentiality queries are performed only for attributes listed in pragmas. Relevant results from the model from subsection 4.1 are presented either as a list of e.g. Reachable States, Non Reachable States, Confiden-

Table 2: Basic block translation rules

$$
\llbracket n,T \rrbracket = \begin{cases}
\bigoplus_{i\in I} \text{"}\textit{new choice}_i\textit{; out choice}_i\text{"} & \\
\bigoplus \text{"}\textit{in choice}\text{"} & \\
\bigoplus_{i\in I} \Big(\text{"}\textit{if choice}=\textit{choice}_i \textit{ then}\text{"} \oplus \llbracket n_i, T' \cup \{t_i\} \rrbracket\Big) & \text{if } T = T' \uplus T'' | T'' = \{t_i \in T | t_i = (n, n_i, g_i, \mathbb{R}_i)\} \quad (a) \\
\text{"}\textit{if } \mathbb{L}(g) \textit{ then}\text{"} \oplus \llbracket n', T \smallsetminus t \cup \{(n, n', \textit{true}, \mathbb{R})\} \rrbracket & \text{if } \exists\, t = (n, n', g, \mathbb{R}) \in T | g \neq \textit{true} \quad (b) \\
\text{"}\textit{let } \mathbb{L}(a) = \mathbb{L}(p) \textit{ in}\text{"} \oplus \llbracket n', T \smallsetminus t \cup \{(n, n', \textit{true}, \mathbb{R}')\} \rrbracket & \text{if } \exists\, t = (n, n', \textit{true}, \mathbb{R}) \in T | \mathbb{R} = (a, p) :: \mathbb{R}' \quad (c) \\
\text{"}\textit{event enteringState } (\mathbb{L}(n'))\text{"} \oplus \llbracket n', T \smallsetminus t \rrbracket & \text{if } \exists\, t = (n, n', \textit{true}, \emptyset) \in T | n' \in S \smallsetminus \textit{Init}_{BB}(b) \quad (d) \\
\text{"}\textit{out (chctrl, tok(call\_}\mathbb{L}(n'), \textit{arg}))\cdot\text{"} & \text{if } \exists\, t = (n, n', \textit{true}, \emptyset) \in T | n' \in \textit{Init}_{BB}(b) \quad (e) \\
\qquad\qquad \text{with } \textit{arg} = \bigcup_{a\in\mathcal{A}} \{\mathbb{L}(a)\} & \\
\text{"}\textit{new } \mathbb{L}(a)\text{"} \oplus \llbracket \nu.a, T \smallsetminus t \rrbracket & \text{if } \exists\, t = (n, \nu.a, \textit{true}, \emptyset) \in T \quad (f) \\
\text{"}\textit{out} (\mathbb{L}(m), \mathbb{L}(p))\text{"} \oplus \llbracket \bar{m}\langle p\rangle, T \smallsetminus t \rrbracket & \text{if } \exists\, t = (n, \bar{m}\langle p\rangle, \textit{true}, \emptyset) \in T \quad (g) \\
\text{"}\textit{in} (\mathbb{L}(m), \mathbb{L}(a))\text{"} \oplus \llbracket m\langle a\rangle, T \smallsetminus t \rrbracket & \text{if } \exists\, t = (n, m\langle p\rangle, \textit{true}, \emptyset) \in T \quad (h) \\
\text{"}.\text{"} & \textit{Otherwise} \quad (i)
\end{cases}
$$

tial Data, Satisfied Authenticity .... For user convenience, the text results are also conveyed on the block and state diagrams, as previously shown in Figure 2 and 3.

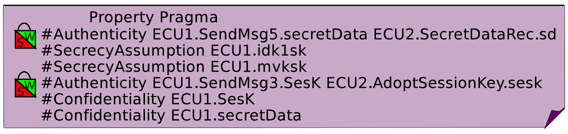
After a ProVerif verification, the state diagram displays the reachability of states, with red locks signifying unreachable states, and green ones signifying reachable ones. The block diagram displays the Authenticity and Confidentiality verification results, shown on pragma as a colored split lock, the bottom left half referring to Strong Authenticity, and the top right half referring to Weak Authenticity. A lock half colored green refers to satisfied authenticity, red to non-satisfied, and grey refers to that which cannot be proved. Attributes proven confidential are displayed with a green lock and those proven non-confidential are displayed with a red lock.

From the asymmetric key distribution described in section 4.1, we exemplify the capabilities and limitations of our translation to ProVerif. The ProVerif verification in Figure 2 shows that confidentiality was verified for the keys *mvksk* and *idk1sk* in both ECU1 and ECU2. Both Strong and Weak Authenticity were proven for the two authenticity queries shown.

In the state diagrams shown in Figure 3, we confirm that states critical to the protocol are reachable, and that the protocol is capable of completion. States such as *UnexpectedMsg6*, which would be reachable only due to unexpected behavior, are confirmed to be unreachable.

Next, we demonstrate that we are capable of performing security verification on state diagrams containing loops. Within ECU2, we add a transition from the last state *ProtocolRunEnded* back to the first state *WaitingForMsg0* to form a loop. No other modifications are made to the block or state diagrams. ECU2 is now ready to pair with any willing entity and should use a different key for each session.

The Confidentiality properties remain identical, but the authenticity properties have changed as shown in Figure 4. Strong Authenticity is no longer verified with the loop because an attacker that would have intercepted the whole exchange between ECU1 and ECU2 would be able to replay it to ECU2. This could be corrected by using nonce instead of timestamps, which are considered constant. Our case study presents an example validation and the changes due to loops.



Property Pragma
#Authenticity ECU1.SendMsg5.secretData ECU2.SecretDataRec.sd
#SecrecyAssumption ECU1.idk1sk
#SecrecyAssumption ECU1.mvksk
#Authenticity ECU1.SendMsg3.SesK ECU2.AdoptSessionKey.sesk
#Confidentiality ECU1.SesK
#Confidentiality ECU1.secretData

Figure 4: Asymmetric Key Distribution Protocol with Loop Authenticity Result

6 CONCLUSION

SysML-Sec targets the full development cycle of a safe and secure embedded system. It is supported by a user-friendly toolkit explicitly supporting safety and security engineering, and offering formal verification at the push of a button. Yet, previous contributions for the proof of security properties had strong limitations both in terms of modeling and proof capabilities. The paper thus proposes a fully novel approach based on new modeling elements (pragmas) and a new model-to-proverif transformation. The paper formalizes both the modeling elements and the transformation. In particular, this paper shows that our approach now supports loops and private channels, and its integration into TTool.

Nonetheless, translating SysML-Sec diagrams to ProVerif still includes limitations. They depend either on ProVerif limitations or on the new translation process. We present them here as a warning to potential users and as leads for potential future work.

Loops. Even though our current translation of loops enables proof of most properties — proof of strong authentication would sometimes fail — and gives sound results for others, the introduction of nonces for chaining basic blocks may in some cases cause ProVerif to produce a *cannot be proved* result. We made this compromise since loops were likely to reduce the completeness of the proof. A possible improvement would be to enable the user to manually provide hints to help ProVerif in its proof for diagrams containing loops.

Arithmetic. When translating the SysML-Sec diagram to ProVerif we discard any arithmetic-related operation that is performed in actions or guards. In fact, ProVerif has no representation for arithmetic, regarding operations or even types like numbers. Taking these operations into account would require us to deeply modify the ProVerif proof engine, such as interfacing it with a theory solver for instance. This is not part of our work in the foreseeable future.

Time. Even though timers are taken into account by our current translation, SysML *after* clauses are not yet handled. However, features were added to ProVerif in order to enable modeling of *phases* which could possibly be used to translate these clauses to some extent.

Other future work include a mathematical proof of equivalence and a design-to-executable_code process preserving security properties.

REFERENCES

Ali, Y., El-Kassas, S., and Mahmoud, M. (2009). A rigorous methodology for security architecture modeling and verification. In *Proceedings of the 42nd Hawaii International Conference on System Sciences*, volume 978-0-7695-3450-3/09. IEEE.

Apvrille, A. (2015). Geek usages for your fitbit flex tracker hack.lu, luxemburg, october 2015. Slides at framadrive.org/index.php/s/Wk6nxAKMpVTdQl4.

Apvrille, L. (2003). TTool. ttool.telecom-paristech.fr.

Apvrille, L. and Roudier, Y. (2015). SysML-Sec: A model driven approach for designing safe and secure systems. In *3rd International Conference on Model-Driven Engineering and Software Development, Special session on Security and Privacy in Model Based Engineering*, France. SCITEPRESS Digital Library.

Assolini, F. (2012). The Tale of One Thousand and One DSL Modems, kaspersky lab.

Bengtsson, J. and Yi., W. (2004). Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*, pages 87–124. W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag.

Blanchet, B. (2009). Automatic Verification of Correspondences for Security Protocols. *Journal of Computer Security*, 17(4):363–434.

Drouineaud, M., Bortin, M., Torrini, P., and Sohr, K. (2004). A first step towards formal verification of security policy properties for rbac. In *QSIC'04*, pages 60–67, Washington, DC, USA.

Durgin, N., Lincoln, P., Mitchell, J., and Scedrov, A. (2004). Multiset rewriting and the complexity of bounded security protocols. *J. Comput. Secur.*, 12(2):247–311.

Hoppe, T., Kiltz, S., and Dittmann, J. (2011). Security Threats to Automotive CAN Networks - Practical Examples and Selected Short-Term Countermeasures. *Rel. Eng. & Sys. Safety*, 96(1):11–25.

ICS-CERT (2015). Hospira lifecare pca infusion system vulnerabilities, advisory (icsa-15-125-01b). https://ics-cert.us-cert.gov/advisories/ICSA-15-125-01B.

Jürjens, J. (2007). Developing secure embedded systems: Pitfalls and how to avoid them. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 182–183. ACM.

Kelling, E., Friedewald, M., Leimbach, T., Menzel, M., Säger, P., Seudié, H., and Weyl, B. (2009). Specification and Evaluation of e-Security Relevant Use cases. Technical Report Deliverable D2.1, EVITA Project.

Maña, A. and Pujol, G. (2008). Towards formal specification of abstract security properties. In *The Third International Conference on Availability, Reliability and Security*, volume 0-7695-3102-4/08. IEEE.

Maslennikov, D. (2010). Russian cybercriminals on the move: profiting from mobile malware. In *The 20th Virus Bulletin Internation Conference*, pages 84–89, Vancouver, Canada.

Maynor, D. (2006). Scada security and terrorism: We're not crying wolf! In *Invited presentation at BlackHat BH 2006. Presentation available at: https://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Maynor-Graham-up.pdf*, USA.

Shen, G., Li, X., Feng, R., Xu, G., Hu, J., and Feng, Z. (2014). An extended uml method for the verification of security protocols. In *Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on*, pages 19–28.

Toussaint, M. J. (1993). A New Method for Analyzing the Security of Cryptographic Protocols. In *Journal on Selected Areas in Communications*, volume 11, No. 5. IEEE.

Trcek, D. and Blazic, B. J. (1995). Formal language for security services base modelling and analysis. In *Elsevier Science Journal, Computer Communications*, volume Vol. 18, No. 12. Elsevier Science.