DOI: http://dx.doi.org/10.14236/ewic/ICS2016.8

Adaptive Modelling for Security Analysis of Networked Control Systems

Jan Wolf, Felix Wiezorek, Frank Schiller
Beckhoff Automation GmbH & Co. KG, Nuremberg, Germany

Gerhard Hansch, Norbert Wiedermann, Martin Hutle
Fraunhofer Institute AISEC, Garching, Germany

Incomplete information about connectivity and functionality of elements of networked control systems is a challenging issue in applying model-based security analysis in practice. This issue can be addressed by modelling techniques providing inherent mechanisms to describe incomplete information. We present and exemplary demonstrate a new, ontology-based method to adaptively model and analyse networked control systems from a security perspective. Our method allows modelling different parts of the system with different levels of detail. We include a formalism to handle incomplete information by applying iterative extension and iterative refinement of the model where necessary. By using machine-based reasoning on an ontology model of the system, security-relevant information is deduced. During this process, non-obvious attack vectors are identified using a structural analysis of the model and by connecting the model to vulnerability information.

decomposition, ontology, threat analysis, system modelling, network security, reasoning, vulnerability

1. INTRODUCTION

Comprehensive security analysis is a fundamental step in securing the Smart Grid. However, due to the complexity of the System Under Evaluation (sue), together with the heterogeneous nature of the Smart Grid, this analysis is a difficult and tedious task. Machine-supported analysis, based on a formal description of the system, is a possibility to cope with this complexity. Hence, recent work on machine-based reasoning like Sommestad et al. (2013); Ji et al. (2009); Ou et al. (2005); Zakeri et al. (2006) introduced some degree of automation to this process.

The key factor for such a model-based analysis is the accuracy and completeness of the underlying model. Achieving accuracy and completeness in practice is often the hardest part of the analysis. The security analyst has to cope with incomplete or inaccurate information about the sue. Moreover, not all parts of the system need to be modelled on the same level of detail.

Current models for ontology-based reasoning require complete information with respect to the language they define and do not allow to have different levels of detail. In contrast, in this paper we provide an ontology to model networked control systems — typical for industrial control systems and the Smart Grid — in a flexible and adaptive way. Our generic ontology language allows modelling systems on different levels of detail. We give an iterative process that allows developing a system description in an adaptive way: The basic elements in our ontology are modules and interfaces. The analyst starts with an initial system template, the initial module, and identifies the (external) interfaces of the system. Then, iteratively, interfaces and modules are expanded, hereby increasing the information about the system and the level of detail. The expansion needs not to be done evenly, that is, some parts of the system can be described more in detail than others.

This approach allows us to focus our modelling on those parts of the system that are interesting for the security analysis. Moreover, comprehensive information about the system is not always available, or is acquired subsequently after a preliminary assessment. Our approach enables such a procedure, as at any stage a security assessment can be conducted. In addition, our modular approach allows reusing already modelled parts.

In this work, we provide an ontology language specification for adaptive modelling and describe its refinement and expansion method. The approach shows its full power once the information gathering process is supported by automatic tools. Hence, we present a short outlook on how standard information security tools can be used to further automate this process. Then we demonstrate the use of the models by applying a machine-based security analysis onto it. Our machine-based reasoning allows the discovery of potential attack vectors. We further illustrate, how already known vulnerabilities documented in various sources can be incorporated and used in the security analysis.

The paper is organised as follows. In Section 2, we introduce the basic language elements for modelling. Further definitions are introduced later in the paper when they become relevant. Section 3 describes the iterative process of model refinement through expansion of interfaces and modules. There, we also provide an outlook on how such a process can be supported by tools. In Section 4, the machine-based deduction of implicit knowledge is addressed. The modelling of vulnerabilities and the linkage to vulnerability databases are described in Section 5. Section 6 describes the automated security analysis based on our ontology language. Related work is given in Section 7. We conclude and present an outlook for further applications of our approach in Section 8.

2. ADAPTIVE ONTOLOGY FOR NETWORKED CONTROL SYSTEMS

2.1. Basic Concepts

An ontology in the formal description language OWL 2 DL is expressed in terms of ‘concepts’, ‘roles’, and ‘individuals’ (see Hitzler (2008)). The membership of an individual $\mathtt{I}$ to a concept C is denoted by $C(\mathtt{I})$. Roles are partial functions on individuals. A role $r(x, y)$ can be interpreted as a relation between individuals x and y, meaning x has a property r to y. We define the domain and range of roles by writing them as partial functions ($r : \langle domain\rangle \nrightarrow \langle range\rangle$). If C_1 is a sub-concept of C_2 (i.e. $\forall x : C_1(x) \rightarrow C_2(x)$), we denote this by $C_1 \sqsubseteq C_2$, and analogue for sub-roles. This hierarchy of concepts is used by a semantic reasoner to infer and extend the model. The operators $\equiv$, $\sqcup$, and $\sqcap$ are concept equality, concept union, and concept intersection, respectively. The concept containing all individuals is denoted by $\top$ while the empty concept is denoted by $\bot$. For a role R and a class C, the class-expression $\exists R.C$ denotes the set of all individuals connected via R to another individual, which is an instance of C. Furthermore, $\forall R.C$ describes the class of all individuals for which all via R related individuals must be instances of C. Number restrictions (like $\exists_{\geq n}$) are used to describe the number of individuals, related to a role. In addition, we give expressions in predicate logic. Variables in predicate logic are written as $?x$.

Expressions about classes allow the machine-based reasoner to deduce implicit knowledge that is not explicitly stated. Assume for instance that a concept $C_1 \sqsubseteq C_2$. Then any individual $\mathtt{I}$ that is in C_1 is by definition also in C_2 and therefore any statement made on the more general concept C_2 holds also for $\mathtt{I}$. In particular, the operations $\equiv$, $\sqcup$, and $\sqcap$ allow construction of new concepts. The reasoner is also able to identify contradictions in the model. The inferring of new roles is addressed in more detail in Section 4.

The basic concepts in our modelling of networked systems are *Interface* and *Module*. Modules characterise entities such as systems, hardware, software, or components. Interfaces provide connection points between modules and comprise hardware interfaces (e.g. connectors), protocol interfaces, or interprocess communication interfaces of an operating system.

We further distinguish interfaces in *Sender* and *Receiver*. These sub-concepts indicate the ability of an *Interface* to send respectively receive data. The actual existence of both abilities is defined as concept *Bidirectional*. A bidirectional interface which is *InitiateOnly* does not accept incoming communication (e.g. a client) and only processes communications it started previously. A bidirectional interface which is *ListenOnly* does not initiate any communication (e.g. a server) but processes incoming communication only. Relations between these sub-concepts are listed below:

$$\begin{aligned} \mathit{Interface} &\equiv \mathit{Sender} \sqcup \mathit{Receiver} \qquad (1)\\ \mathit{Bidirectional} &\equiv \mathit{Sender} \sqcap \mathit{Receiver} \\ \mathit{ListenOnly} &\sqsubseteq \mathit{Bidirectional} \\ \mathit{InitiateOnly} &\sqsubseteq \mathit{Bidirectional} \\ \bot &\equiv \mathit{InitiateOnly} \sqcap \mathit{ListenOnly} \end{aligned}$$

The following basic roles are defined for interfaces and modules:

- *hasInterface* : *Module* ⇸ *Interface* assigns interfaces to modules.
- *connected* : *Sender* ⇸ *Receiver* indicates the possibility of two interfaces to communicate (with respect to connection requiring compatibility).
- *communicates* : *Sender* ⇸ *Receiver* indicates that two interfaces are communicating.
- *compatible* : *Sender* ⇸ *Receiver* indicates that two interfaces are compatible

Formally, these roles are defined by specifying the domain and the range of each role:

$$\begin{aligned} \mathit{Interface} &\sqsubseteq \forall\, \mathit{hasInterface}.\mathit{Module} \qquad (2)\\ \exists\, \mathit{hasInterface}.\top &\sqsubseteq \mathit{Module} \end{aligned}$$

For the sake of brevity, we will omit such definitions for the remainder of the paper and provide only the partial function specification.

2.2. Compatibility of Interfaces

Instances of the concept *Interface* can only communicate if they are ‘compatible’. To model this fact, compatible interfaces have to be derived from classes that are defined compatible. This is achieved by predicate logic expressions that are equivalent to Semantic Web Rule Language (swrl) rules. For each compatibility, we provide such a rule. Intuitively, these rules can be regarded as the communication protocols supported by the interfaces:

Example 1 *In order to model the compatibility between a HttpServer and a HttpClient, the following expressions are added to the model:*

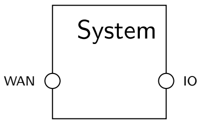
$$\mathit{HttpServer} \sqsubseteq \mathit{ListenOnly}$$
$$\mathit{HttpClient} \sqsubseteq \mathit{InitiateOnly}$$
$$\mathit{HttpServer}(?i_1) \wedge \mathit{HttpClient}(?i_2) \rightarrow \mathit{compatible}(?i_1, ?i_2) \qquad \triangle$$

Only compatible interfaces can be connected and only connected interfaces can communicate:

$$\mathit{communicates} \sqsubseteq \mathit{connected} \sqsubseteq \mathit{compatible} \tag{3}$$

In order to illustrate the model refinement process presented in this work, we use a system as depicted in Example 2 as a running example.

Example 2 *A high level model of an automation system:*



$$\mathit{Module}(\texttt{System}) \wedge \mathit{hasInterface}(\texttt{System}, \texttt{WAN}) \wedge \mathit{hasInterface}(\texttt{System}, \texttt{IO}) \wedge \mathit{Bidirectional}(\texttt{WAN}) \wedge \mathit{Bidirectional}(\texttt{IO}) \qquad \triangle$$

2.3. Security Relations

Instances of *Interface* can be secured, i.e., protected against eavesdropping and manipulation. A secured *Interface* is member of the concept *Secured*.

Key material is a member of the concept *Key*. The following roles describe the linkage between keys and secured interfaces:

- *hasKey* : *Module* $\nrightarrow$ *Key* denotes which modules have knowledge of a certain key.
- *acceptsKey* : *Interface* $\nrightarrow$ *Key* denotes that an secured *Interface* accepts a key to access the connection.

2.4. Functional Dependencies and Access

The outputs of a module depend on the values it receives. In Example 2, if the output `IO` of System could result as a function based on received values on `WAN`, such a dependency would exist (which normally should not be the case). An attacker could then use this property to manipulate the output of a module. We capture these input/output dependencies in a generalised way by the following role:

- *depends* : *Interface* $\nrightarrow$ *Interface* denotes a dependency between two interfaces, i.e., the input at the first interface determines in some way the output at the second interface.

However, the presented role is more general than in the example above. In particular, we can state that any interface which uses another interface (e.g. a network stack) depends on that interface:

$$\mathit{usesinterface}(?i_1, ?i_2) \wedge \neg\, \mathit{Secured}(?i_1) \rightarrow \mathit{depends}(?i_1, ?i_2) \tag{4}$$

Further, any communication implies a dependency between the communicating interfaces. Thus, *communicates* is a sub-role of *depends*:

$$\mathit{communicates} \sqsubseteq \mathit{depends} \tag{5}$$

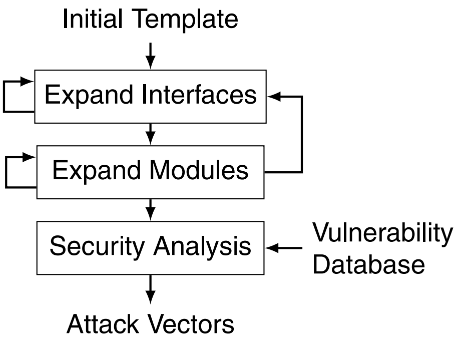
In some cases interfaces allow the modification of the functionality of the associated module. This is, for example, the case for programming interfaces of Remote Terminal Units (rtus) or for remote access interfaces. Without any further specification, we assume by default the worst case that access to an interface allows (the user or an attacker) to fully control the module it belongs to. If this assumption does not apply interfaces are added to the concept *NotControllable*.

$$\mathit{NotControllable} \sqsubseteq \mathit{Interface} \tag{6}$$

3. ITERATIVE MODULE AND INTERFACE REFINEMENT

In order to support a progressive modelling of a system, we apply a hierarchical approach. Initially, we consider the whole sue as a single module, and identify the interfaces of this module. In the final model, these are the *external interfaces*. Example 2 shows how such a modelling might look.

Then, in an iterative process, interfaces and modules are expanded to generate a refined model. This approach allows a model to be more fine granular at sensitive places, and more coarse grained where only limited information is available. When the system is sufficiently modelled, a security analysis (which in our case uses additionally a vulnerability database as input) can be performed. The identification of sensitive places and the sufficiency of the modelling process is subject to security best practices and has to be part of future work. The technique also allows continuation of the model refinement process after a preliminary security analysis. The following figure shows the overall model refinement process.



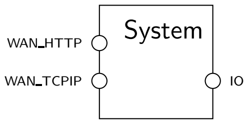
3.1. Expansion of Interfaces

When refining a part of the system, usually modules and interfaces are expanded. In general, it is recommended to expand interfaces prior to their providing modules. This process is described in the following steps .

Step 1: Add contained sub-interfaces
The first step of expansion is to add new interfaces to the module and to relate them to the original one with an *originatesFromI* role.

- *originatesFromI* : *Interface* ⇸ *Interface* denotes that the first *Interface* results from an expansion of the second *Interface*.
- *usesInterface* : *Interface* ⇸ *Interface* denotes that the first *Interface* uses the second *Interface* to implement the communication. This is used e.g. to model network stacks.

Example 3 *An examination of the Interface* `WAN` *in Example 2 shows that the interface contains an HTTP connection using a TCP/IP stack. Therefore, we expand this Interface as depicted below.*



$$\begin{aligned}&\mathit{originatesFromI}(\texttt{WAN\_HTTP},\texttt{WAN})\\ &\wedge\ \mathit{originatesFromI}(\texttt{WAN\_TCPIP},\texttt{WAN})\\ &\wedge\ \mathit{hasInterface}(\texttt{System},\texttt{WAN\_HTTP})\\ &\wedge\ \mathit{hasInterface}(\texttt{System},\texttt{WAN\_TCPIP})\\ &\wedge\ \mathit{usesInterface}(\texttt{WAN\_HTTP},\texttt{WAN\_TCPIP})\end{aligned}$$

Note that the Interface `WAN` *remains part of the knowledge base, although not depicted in the figure above.* $\triangle$

Expanded interfaces, i.e., those for which an *originatesFromI* role exists, are hidden in the network view of the system but remain in the ontology. In an interface expansion step, each sub-interface is provided by the same module as the interface it originates from.

Step 2: Determine if the new interfaces allow access
If the original interface is not controllable, i.e., access to this interface does not allow controlling the associated module, all expanded interfaces are also not controllable. Otherwise, each new interface needs to be assessed if it is part of the concept *NotControllable*.

Example 4 *Based on Example 3 the new interfaces are considered. Without a vulnerability present, the System is not controllable from neither the* `WAN_HTTP` *nor the* `WAN_TCPIP` *interface:*

$$\mathit{NotControllable}(\texttt{WAN\_HTTP}) \wedge \mathit{NotControllable}(\texttt{WAN\_TCPIP})$$

$\triangle$

Step 3: Connect existing communication to new sub-interfaces
If there was a *communicates* or *connected* role to the original interface, we examine for each new interface if the role needs to be copied. Naturally, if there was a *communicates*, there should be at least one interface in the expanded model that also has a *communicates* role with the peer of the original interface. The same procedure applies for all other roles.

Continuing the running example, where the external interface `WAN` is not yet connected, communicate connections an attacker might use are introduced in Example 5.

Example 5 *For a security analysis we assume, that the attacker can access both interfaces, so communicates*$(\texttt{Attacker},\texttt{WAN})$ *is already part of the model (cf. Section 6). The following roles are therefore added to the model:*

$$\begin{aligned}&\mathit{communicates}(\texttt{Attacker},\texttt{WAN\_HTTP})\wedge\\ &\mathit{communicates}(\texttt{Attacker},\texttt{WAN\_TCPIP})\end{aligned}$$

$\triangle$

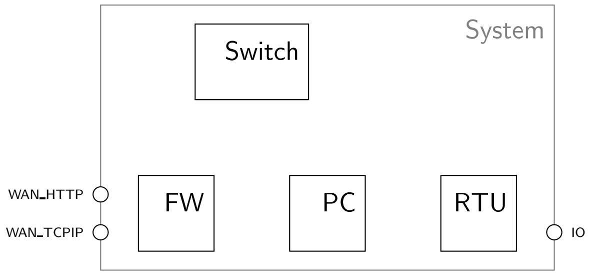
3.2. Expansion of Modules

To allow a more detailed analysis of a system, modules can be partitioned into sub-modules. The link between a module and its sub-modules is described using the *originatesFromM* role:

- *originatesFromM* : $\mathit{Module} \nrightarrow \mathit{Module}$ denotes that the first module results from an expansion of the second module.

Step 1: Add contained sub-modules
When expanding a module, all sub-modules are added to the knowledge base and connected with the *originatesFromM* role. In this step, it is important that the list of sub-modules is complete, i.e., that indeed the sum of all sub-modules constitute the module. In case there are parts that are not supposed to be modelled, a "remainder" module can be used. Expanding the module System presented in Example 3 reveals several sub-modules depicted in Example 6.

Example 6 *An examination of the network plan shows that the System comprises a PC that is used for SCADA and a rtu; in addition there is a firewall (FW) and a Switch.*



$$\begin{aligned}&\mathit{originatesFromM}(\mathtt{FW},\mathtt{System})\\ \wedge\ &\mathit{originatesFromM}(\mathtt{Switch},\mathtt{System})\\ \wedge\ &\mathit{originatesFromM}(\mathtt{PC},\mathtt{System})\\ \wedge\ &\mathit{originatesFromM}(\mathtt{RTU},\mathtt{System})\end{aligned}$$

△

Step 2: Assign external interfaces
Using the *hasInterface* role, the original interfaces are assigned to the expanded modules. Each interface of the original module must be assigned to exactly one module in the expanded model. In Example 7 this is performed based on Example 6.

Example 7 *We analyse our system and locate the TCP/IP interface at the firewall and the IO interface at the rtu. The HTTP connections are transparently routed over the firewall, thus this interface is at the PC:*

$$\begin{aligned}&\mathit{hasInterface}(\mathtt{FW},\mathtt{WAN\_TCPIP}) \wedge \mathit{hasInterface}(\mathtt{PC},\mathtt{WAN\_HTTP})\\ \wedge\ &\mathit{hasInterface}(\mathtt{RTU},\mathtt{IO})\end{aligned}$$

△

Note that the assignment to the outer entity (in our example the System) remains.

Step 3: Create internal interfaces
All internal interfaces are added to the model. They are attached using *hasInterface* to the newly created modules and added to the proper interface sub-concepts in order to provide information about compatibility of interfaces. There is no need for *originatesFromX* roles for these new interfaces as their origin can be determined by the modules they are attached to by *hasInterface*. In Example 8, internal interfaces are integrated to the model.

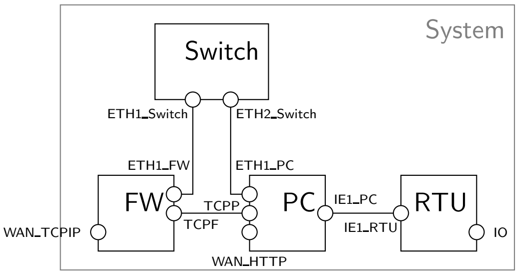
Example 8 *The following internal interfaces are added to the model: internal Ethernet, TCP/IP, and Industrial Ethernet.*

$$\begin{aligned}&\mathit{hasInterface}(\mathtt{FW},\mathtt{ETH1\_FW}) \wedge \mathit{EthInterface}(\mathtt{ETH1\_FW})\\ &\mathit{hasInterface}(\mathtt{FW},\mathtt{TCPF}) \wedge \mathit{TcpIpInterface}(\mathtt{TCPF})\\ &\mathit{hasInterface}(\mathtt{Switch},\mathtt{ETH1\_Switch}) \wedge \mathit{EthInterface}(\mathtt{ETH1\_Switch})\\ &\mathit{hasInterface}(\mathtt{Switch},\mathtt{ETH2\_Switch}) \wedge \mathit{EthInterface}(\mathtt{ETH2\_Switch})\\ &\mathit{hasInterface}(\mathtt{PC},\mathtt{ETH1\_PC}) \wedge \mathit{EthInterface}(\mathtt{ETH1\_PC})\\ &\mathit{hasInterface}(\mathtt{PC},\mathtt{TCPP}) \wedge \mathit{TcpIpInterface}(\mathtt{TCPP})\\ &\mathit{hasInterface}(\mathtt{PC},\mathtt{IE1\_PC}) \wedge \mathit{IndustrialEthernet}(\mathtt{IE1\_PC})\\ &\mathit{hasInterface}(\mathtt{RTU},\mathtt{IE1\_RTU}) \wedge \mathit{IndustrialEthernet}(\mathtt{IE1\_RTU})\end{aligned}$$

△

Step 4: Communication between internal interfaces
Finally, the newly created interfaces are linked using the *communicates* role (in case the components are supposed to communicate) and the *connected* role (if they are connected, it is possible that they could communicate). This step is illustrated in Example 9.

Example 9 *All linked interfaces communicate with their counterparts:*



For each pair of interfaces that are linked we add a communicates role:

$$\begin{aligned}&\mathit{communicates}(\mathtt{ETH1\_FW},\mathtt{ETH1\_Switch})\\ &\ldots\end{aligned}$$

Note that the connected role follows implicitly from (3). △

Step 5: Add access information
Similar to Step 2 of the expansion of interfaces (see Section 3.1), it needs to be assessed if the new interfaces are in the concept *NotControllable*.

Step 6: Add functional dependencies
For all new interfaces not connected with a *communicates* role (for which the dependency role is given implicitly), the *depends* role needs to be examined.

Example 10 *In the expanded system of Example 9, we identify that all interfaces except the* `IE1_RTU` *are in NotControllable. Further, there are functional dependencies inside the FW and the rtu, as depicted in the figure in Example 13.* △

3.3. Semantical information gathering

In addition to the structural expansion of modules and interfaces, we use sub-concepts to represent additional information about the module respectively interface. The purpose is twofold: first, only by specifying subtypes, can specific knowledge necessary for a non-trivial security analysis be collected. Second, the recognition and reuse of previously refined models is possible.

3.3.1. Typing of Sub-models
When expanding interfaces and modules, the new individual should be added to a concept that captures as much information about the individual as possible. This is illustrated based on a PC and its operating system in the following Example 11.

Example 11 *A new module* `m` *is a Windows PC, and we know that it is a Windows 7. We associate Windows with* `m`*, and Windows7 as a sub-concept of Windows, and state Windows7(*`m`*).* △

3.3.2. Recognition and Reuse of Sub-models
A detailed sub-concept hierarchy enables the collection of a template library for already modelled systems. Subsequent to Example 11, we can reuse the full expanded model as a template for other PCs running Windows 7.

3.4. Tool-based Expansion

As stated initially, the model expansion steps can be performed manually but show their full power when combined with tools generating this information automatically. Security tools for information gathering are good candidates to provide such information. We illustrate this approach by showing how a generic TCP/IP interface of a system can be expanded into a set of interfaces that are actually present in the system:

`nmap` is a tool commonly used to scan a host for open ports (see Lyon (2008)). For instance, we could obtain triples $(\mathit{protocol}, \mathit{port}, \mathit{status})$ when scanning the IP interface of a specific host, where *protocol* is from $\{\mathtt{tcp}, \mathtt{udp}, ...\}$, *port* is a number between 0 and 2^{16}, and *status* is $\{\mathtt{open}, \mathtt{closed}, \mathtt{filtered}\}$.

Let $?m$ be a module of a network that we want to expand. In a first step we query in swrl for all IP interfaces of this module:

$$\mathit{hasInterface}(?m, ?i) \wedge \mathit{IpInterface}(?i)$$
$$\rightarrow \mathit{sqwrl{:}select}(?i)$$

Then let `IF4711` be one of these interfaces where `nmap` found an open TCP/80 port. By adding this to our knowledge base we expand `IF4711`:

$$\mathit{TCP80Interface}(\mathtt{IF4711TCP80}) \wedge$$
$$\mathit{originatesFromI}(\mathtt{IF4711TCP80}, \mathtt{IF4711})$$

4. INFERRING ABOUT MODELS

A major advantage of ontological modelling is that in addition to the explicitly gathered information, further correlations can be expressed by adding rules. These often simple rules can be evaluated by a reasoner. Even in large systems, where it would be difficult to keep track of consequences if done manually, a reasoner can evaluate queries such as consistency.

To represent this inferable knowledge, adequate rules need to be added. Some important rules are presented in this section.

4.1. Inferring *connected* Relations

An important issue for security analysis is to determine the interconnection of a system. Even designers are often not aware of all connections in complex systems.

A starting point when inferring further connections is that two interfaces are connected if

- they are compatible and
- the base interfaces they are using are connected and
- they are either not secured, or the key is shared.

These points are formalised by the statement below:

$$\begin{aligned}
&\mathit{hasInterface}(?m_1, ?i_1) \wedge \mathit{hasInterface}(?m_2, ?i_2) \wedge \\
&\mathit{usesInterface}(?i_1, bi_1) \wedge \mathit{usesInterface}(?i_2, ?bi_2) \wedge \\
&\mathit{connected}(?bi_1, ?bi_2) \wedge \mathit{compatible}(?i_1, ?i_2) \wedge \\
&(\\
&\quad \neg\, \mathit{Secured}(?i_1) \wedge \neg\, \mathit{Secured}(?i_2) \vee \\
&\quad \mathit{Secured}(?i_1) \wedge \mathit{Secured}(?i_2) \wedge \\
&\qquad (\mathit{hasKey}(?m_1, ?k_1) \wedge \mathit{acceptsKey}(?i_2, ?k_1) \vee \\
&\qquad \mathit{hasKey}(?m_2, ?k_2) \wedge \mathit{acceptsKey}(?i_1, ?k_2)) \\
&) \\
&\rightarrow \mathit{connected}(?i_1, ?i_2) \qquad (7)
\end{aligned}$$

4.2. Inferring *communicates* Relations

It is worthwhile to examine *communicates* roles as these represent actual communication in the sue. The addition of these roles allows, for example, to identify communication where none should occur according to the design of the system (e.g. 'air-gapped' systems).

If two interfaces communicate and each uses only one base interface, then the base interfaces are communicating. This is expressed by the statement below:

$$\begin{aligned}&\textit{communicates}(?i_1, ?i_2)\wedge \\ &\textit{usesInterface}(?i_1, ?bi_1) \wedge (=1)\textit{usesInterface}(?i_1)\wedge \\ &\textit{usesInterface}(?i_2, ?bi_2) \wedge (=1)\textit{usesInterface}(?i_2) \\ &\rightarrow \textit{communicates}(?bi_1, ?bi_2)\end{aligned} \tag{8}$$

If an interface uses more than one base interface, more specific rules (e.g. protocol related) can be included.

5. VULNERABILITY MODELLING

Vulnerabilities describe the system's behaviour (more specifically interfaces) with respect to attacks. Some assumed properties are no longer in the presence of vulnerabilities. We introduce a new concept *Vulnerability* for this. For example, privilege escalation vulnerabilities may provide control through interfaces despite being modelled as *NotControllable*.

Vulnerabilities are characterised by the kind of affected interfaces and the effects on them. In order to describe this, we introduce a new role

- *isVulnerable* : *Interface* ⇸ *Vulnerability* denoting that an *Interface* is vulnerable to a *Vulnerability*.

5.1. Vulnerability Classes

Attacks can either be enabled via inadequate configuration or via vulnerabilities in interfaces. We propose to consider the following three concepts of vulnerabilities:

CodeExecution
An *Interface* modelled to be *NotControllable* allows a manipulation of the *Module* of the *Interface* by using this kind of vulnerability.

For example, arbitrary code execution vulnerabilities such as CVE-2014-6271 (Shellshock) are in this class.

CryptoIneffectiveness
An *Interface* modelled to be *Secured* allows a communication with interfaces belonging to a *Module* regardless of the corresponding *hasKey* Relation.

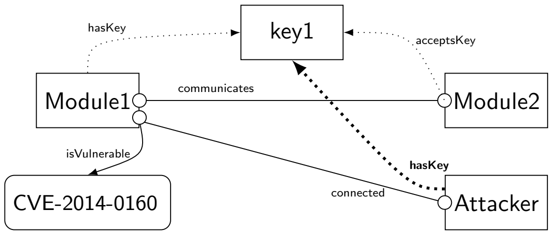
For example, weaknesses in cryptographic algorithms, such as the man-in-the-middle downgrade of TLS to weak export ciphers described by Adrian et al. (2015), are in this class.

InformationDisclosure
Confidential information modelled to be known only to the module, such as keys, is assumed to be extractable through an *Interface* by this kind of vulnerability.

This vulnerability class manifests often in weak passwords or exposition of secret internal states, such as improper input validation like CVE-2014-0160 (Heartbleed).

The following Example 12 visualises this vulnerability.

Example 12 *The vulnerable interface leaks the key to the attacker.*



The Attacker could use the obtained hasKey to communicate to Module2 after exploiting this vulnerability. △

All described Vulnerability classes are sub-concepts of Vulnerability:

$$\begin{aligned}\textit{CodeExecution} &\sqsubseteq \textit{Vulnerability}\\ \textit{CryptoIneffectiveness} &\sqsubseteq \textit{Vulnerability}\\ \textit{InformationDisclosure} &\sqsubseteq \textit{Vulnerability}\end{aligned}$$

6. SECURITY ANALYSIS

A security analysis is possible after each step of model expansion. In the beginning of the expansion process the results will be more generic and will make conservative security assumptions. In order to achieve detailed results, more information has to be added by selectively expanding the model. The results of the analysis can point to parts of the model, which may need further expansion in order to obtain more useful results. It is assumed that attacks only propagate through defined properties (see (Section 6.2).

6.1. Assign Publicly Known Vulnerabilities

As an extension of the modelling it might be possible to assign known vulnerabilities. There exist many services capable of gathering and distributing information about vulnerabilities, e.g. Jajodia et al. (2011). The information from these sources can be aggregated, converted, and put into the model to obtain a more detailed and up-to-date analysis. As a basis, the modules have to be identifiable types of software or hardware such as Common Platform Enumeration (cpe). For the cpe, all known vulnerabilities and their Common Vulnerabilities and Exposures (cve) can be queried from the services. These vulnerabilities can then be mapped to the affected interfaces of the modules. The vulnerabilities then have to be classified for further analysis. This can be done, for example, by mapping the Common Weakness Enumeration (cwe) contained in the cve to vulnerability classes (Martin et al. (2005)).

6.2. Identifying Attack Vectors

The model including vulnerabilities is a directed graph. In a first step to identify attack vectors, the target (T) and the start point (A) of the attack have to be defined.

An edge from $?m$ to $?i$ if is added to the attack graph if the following condition holds:

- *hasInterface*$(?m, ?i)$. This represents the fact that control of a module allows controlling the associated interfaces.

An edge from $?i$ to $?m$ is added if one of the following conditions hold:

- *hasInterface*$(?m, ?i) \wedge \neg$*NotControllable*$(?m)$.
  This reflects the case that the interface (potentially) allows controlling the module it belongs to.
- *hasInterface*$(?m, ?i) \wedge$ *NotControllable*$(?m) \wedge$ *isVulnerable*$(?i, ?v) \wedge$ *CodeExecution*$(?v)$.
  This reflects the case where the interface does not allow controlling the module by design, but is used to gain access by a code execution vulnerability.

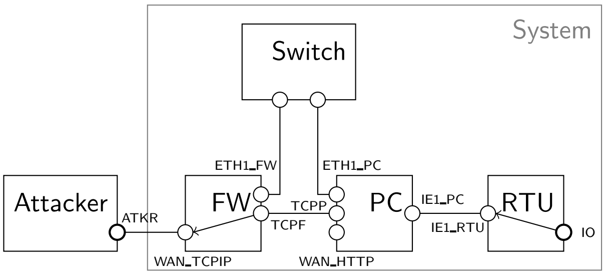
An edge from $?i1$ to $?i2$ is added if one of the following conditions hold:

- *connected*$(?i1, ?i2)$. This represents the fact that access to an interface allows accessing the connected interface.
- *usesInterface*$(?i_1, b_1) \wedge$ *usesInterface*$(?i_2, ?b_2) \wedge$ *connected*$(?b_1, ?b_2) \wedge$ *compatible*$(?i_1, ?i_2) \wedge$ *CryptoIneffectiveness*$(?v) \wedge$ *isVulnerable*$(?i_2, ?v)$
  This reflects the case where a communication is protected by an ineffective method, and circumvented by an attacker that has access to the lower level media.
- *depends*$(?i1, ?i2)$. This represents the situation where the manipulation of an interface leads to changes in a functionally dependent interface.

The single paths of the attack graph are the attack vectors. It is possible to extend this with rules for information disclosure of key material. This leads to complex attack vectors including *and* and *or* conditions, and is omitted for sake of brevity.

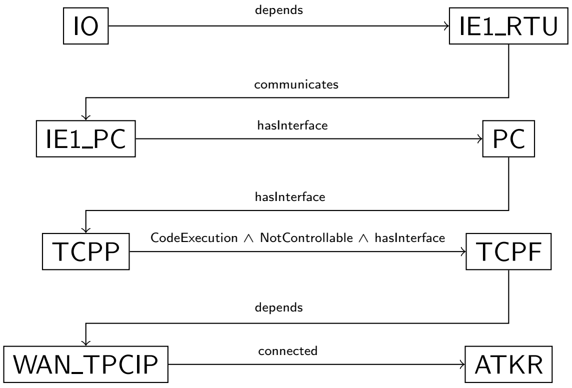
In Example 13, the running example is used to illustrate a security analysis.

Example 13 *The extraction of one specific attack path is depicted based on an augmented Example 9.*



NotControllable$(\mathit{TCPP}) \wedge$ *NotControllable*$(\mathit{ETH1\_PC}) \wedge$
NotControllable$(\mathit{IE1\_RTU}) \wedge$ *NotControllable*$(\mathit{WAN\_TCPIP}) \wedge$
CodeExecution(TCPP)

One resulting attack path can be described as follows. The labels of the edges are the matching terms from above.



△

7. RELATED WORK

Description logic based reasoning is a promising approach to efficiently identify vulnerabilities and security gaps in modelled systems. Ji et al. (2009); Zakeri et al. (2006); Ou et al. (2005) prove the applicability of this method for security analysis. While their work focusses on the analysis of the model, there is no special focus on constructing the ontology model for the analysis and might not be sufficiently comprehensive for a practical, conclusive analysis of complex systems. Security-focussed ontologies are presented by Kim et al. (2005); Fenz and Ekelhart (2009); Fitzgerald (2010). In order to infer models from such ontologies and reason about their elements, semantic reasoners, as described by Glimm et al. (2014); Sirin et al. (2007), are the method of choice to ensure an efficient and formal correct proceeding.

As previously pointed out, the quality of a model based analysis is directly dependent on the quality of the used model. The most prevalent and widespread modelling language in the field of computer science is the Unified Modeling Language (uml), which is, however, not designed for automated, machine-based security analysis. Specific modelling elements are provided as language extensions, like for security SecureUML by Lodderstedt et al. (2002) and UMLSec by Jürjens (2002), or complete dialects such as SysML (which focusses on support of the specification, analysis, design, verification and validation of systems and systems-of-systems). While these model languages are effective during time of design, they lack methods to break down or expand single components and describe object relations. Detailed knowledge about the critical components, their parts, and relations is however crucial for the analysis. None of the methods allow an ontology-based completion process by using patterns of known elements, ontological knowledge, and a semantic reasoner.

The Cyber Security Modelling Language (CySeMol) by Sommestad et al. (2013) and its extension by Holm et al. (2015) combine UML-based information system modelling with Bayesian attack graphs to assess attack probabilities for a modelled system. The use of the relational model and the thereupon built inference engine allows the far-reaching evaluation of 'what-if' scenarios. Networks consisting of well-known components can be evaluated efficiently due to the predefined granularity of the components. While this approach enables modifications of the model during analysis, it does not include iterative dissection, refinement or a way to model a lack of knowledge about the components of the system.

When searching for vulnerabilities of modules, it is essential to have links to their corresponding entries at vulnerability databases (like NIST NVD or Bugtraq). Promising connectors to such lists of cves are the cpe or the cwe. cpe is a standardised method of describing and identifying classes of applications, operating systems, and hardware devices present among an enterprise computing system. cwe is a list of software weaknesses that might result in a vulnerability of a product. These lists can be used in combination with previous investigations. The creation and linking of such databases to assign weaknesses, idiosyncrasies, faults and flaws (WIFFs), described by Martin et al. (2005), enables various forms of automated security analysis and penetration reports like e.g. Knorr et al. (2011).

Tools to perform an attack graph based security analysis are described by Ou et al. (2005); Lippmann et al. (2006); Noel et al. (2009); Jajodia et al. (2011). It is typical for these approaches to separate the system model generation from the model analysis. The input, which is required to create the model, is gathered by a network security scanner (like Nessus) and combined with some topology information in a predefined class model, especially information on how devices are interconnected. During the analysis of the model, vulnerability databases are searched for known vulnerabilities of the modelled modules while the initially created model itself remains unmodified.

To the best of our knowledge, all the works described above do not address incomplete or varying granularity of knowledge about the parts of the sue nor how to refine the initial model as part of the analysis. The created model is considered as a complete and correct basis for the entire analysis. Modifications are only used to evaluate effects of augmenting or removing modules. Our approach, in contrast, uses a reasoning engine not only to evaluate to, but also to refine and expand the model with ontological knowledge as well as predefined patterns.

8. CONCLUSION

In this paper, an modelling approach for machine-based security analysis using ontologies is presented. We propose a process of selective iterative model refinement, which allows different levels of detail for different parts of the system. For this, we provide an ontology language that allows us to formally describe the system and to infer implicit knowledge. We illustrate the applicability of the resulting model for security analysis by combining the model with vulnerability information. Our security analysis reveals attack vectors resulting from paths through the system and from exploiting vulnerabilities.

The approach is very generic and further applications can be considered. For example, the analysis can be enriched by applying probability values to the vulnerability classes and using a probability analysis (like in Bayesian networks) to find the most relevant attack vectors. For large systems and more detailed analysis, more tool support is mandatory and under current development.

ACKNOWLEDGEMENT

The authors are grateful to the Bavarian Ministry of Economic Affairs and Media, Energy, and Technology for supporting this work by funding the project SustainGrid (IuK423) within the research programme Information and Communication Technology.

REFERENCES

Adrian, D. *et al.* (2015). Imperfect forward secrecy: How Diffie–Hellman fails in practice. In: *22nd ACM Conference on Computer and Communications Security*, 5–17.

Fenz, S. and A. Ekelhart. (2009). Formalizing information security knowledge. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, 183–194.

Fitzgerald, W. M. (2010). An ontology engineering approach to network access control configuration.

Glimm, B. et al. (2014). HermiT: An OWL 2 Reasoner. *Journal of Automated Reasoning*, 53(3), 245–269.

Hitzler, P. (2008). *Semantic Web: Grundlagen*. Berlin, Germany: Springer-Verlag.

Holm, H. et al. (2015). P^{2}CySeMoL: Predictive, Probabilistic Cyber Security Modeling Language. *IEEE Transactions on Dependable and Secure Computing*, 12(6), 626–639.

Jajodia, S. et al. (2011). Cauldron mission-centric cyber situational awareness with defense in depth. In: *Military Communications Conference*, 1339–1344.

Ji, Y. et al. (2009, Aug.) A logic-based approach to network security risk assessment. In: *Computing, Communication, Control, and Management, 2009. CCCM 2009. ISECS International Colloquium on*, 3, 9–14.

Jürjens, J., Ed. (2002). *UMLsec: Extending UML for Secure Systems Development*. Berlin, Germany: Springer.

Kim, A. et al. (2005). *Security Ontology for Annotating Resources*. Springer.

Knorr, K. et al. (2011). Automatisierung von Penetrationstest-Berichten mittels CWE. *DACH Security 2011*.

Lippmann, R. et al. (2006). Validating and Restoring Defense in Depth Using Attack Graphs. In: *IEEE Military Communications conference*, 1–10.

Lodderstedt, T. et al., Eds. (2002). *SecureUML: A UML-Based Modeling Language for Model-Driven Security*. Berlin, Germany: Springer.

Lyon, G. F. (2008). *Nmap Network Scanning - Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure.Com, LLC.

Martin, R. A. et al. (2005). The case for common flaw enumeration. In: *NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics*.

Noel, S. et al. (2009). Advances in topological vulnerability analysis. In: Conference For Homeland Security, 2009. CATCH '09. Cybersecurity Applications & Technology, 124–129.

Ou, X. et al. (2005). MulVAL: A logic-based network security analyzer. In: *USENIX Security*, 8.

Sirin, E. et al. (2007). Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web* 5(2), 51–53.

Sommestad, T. et al. (2013). The cyber security modeling language: A tool for assessing the vulnerability of enterprise system architectures. *IEEE Systems Journal*, 7(3), 363–373.

Zakeri, R. et al. (2006). Using description logics for network vulnerability analysis. In: *Fifth International Conference on Networking and the International Conference on Systems (ICN/ICONS/MCL*, Mauritius, 78.