Analysis of the SYM2 Smart Meter Remote Software Download using formal methods reasoning

Andreas Fuchs
Fraunhofer Institute for Secure Information Technology
Rheinstrasse 75
Darmstadt, Germany
andreas.fuchs@sit.fraunhofer.de

Donatus Weber
Chair for Data Communications Systems
University of Siegen
Hölderlinstrasse 3
Siegen, Germany
donatus.weber@uni-siegen.de

ABSTRACT

The extended use of clean electricity in the future requires an intelligent distribution network, the so called Smart Grid. A main component of this grid are Smart Meters, which are capable of providing services like the remote readout of the actual power consumption or the remote control of loads.
Modern Smart Meters are Embedded Systems, running software on a microcontroller platform. Due to adding new features or fixing errors, it is necessary to remotely update the software of Smart Meters in the field. Since Security plays a major role for critical infrastructure components like Smart Meters, a Secure Software Download mechanism is strongly needed. The German SYM^2 specification for Smart Meters proposes variants for the Secure Software Download of the legally relevant modules as well as for the non-legally relevant modules.
Both variants will be formally investigated and compared in regard to provided security properties using the Fraunhofer Secure Modeling Framework (SeMF). The real world system is mapped to a system model comprising all necessary actions, requirements and assumptions. It serves as a basis for the formal method reasoning. Security issues are pointed out and analysed towards possibilities for secure solutions.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements Specifications—*Methodologies*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*

General Terms

Security, Legal Aspects

Keywords

Smart Metering, Formal Methods, Security Requirements

S&D4RCES '11 September 22, 2011, Naples, Italy.

1. INTRODUCTION

During the last years, the upcoming importance of clean energy and legal regulations in the energy sector forced a growing need for an intelligent, dynamically manageable power grid (Smart Grid). For real-time load managing and the use of wind and solar power, the Smart Grid needs periodically updated information about the actual power consumption of all connected households and industries. This leads to a growing need for Smart Meters in the electricity sector.

The Smart Electricity Meter is equipped with a communication unit enabling the data transfer to the Remote Readout Center or Measuring Point Operator over different kinds of networks. These networks range from internet connections using the TCP/IP protocol over GSM/UMTS to built-for-purpose Power Line Communication (PLC) techniques. For the Network Operators as well as for the Energy Suppliers, it is of great importance to have updated values of the current grid load. Typical time constants for the periodical readout of Smart Electricity Meters are 15 minutes. These periodic readouts serve the Energy Suppliers as a basis to balance loads in the grid, helping to avoid peaks in demand.

Beside the functionality of providing periodic readouts of the actual power consumption, modern Smart Meters offer the opportunity to remotely switch loads in households and industry. This service is needed for the Smart Grid to efficiently use clean energy at the time it is available. In order to be able to offer the required functionalities, an analogue, electromechanical setup for a meter is not sufficient any more. Modern Smart Meters are resource-constrained embedded systems running different kinds of software. Typical constraints are computation speed, memory size and energy.

Furthermore, the Smart Electricity Meter is an important component of the critical energy supplying infrastructure. Security flaws in Smart Meters offer attackers a good opportunity to manipulate data on the Smart Grid.

This effort may even result in financial profit. Manipulating the actual consumption information for a larger number of Smart Meters can directly influence prices in the energy exchange. Because of these potential risks, security engineering is an important component in the development process of a Meter.

An interesting approach for harmonising the Smart Electricity Meter domain is the SYM^2 specification [12] pub-

lished by the German tLZ working group. SYM^2 is the synonym for *Synchronous Modular Meter*, a modular and non monolithic concept for load profile meters with clocked registration periods.

As Smart Meters run software, it may become necessary to download a new software version on the Smart Meter for fixing errors or adding new features. For providing this functionality, the SYM^2 specification offers Remote Software Download mechanisms. Work within this paper concentrates on the investigation of the procedure in regard to security. Basis for the formal investigation is the equally young Secure Modeling Framework. Its methods and concepts are going to be applied to Smart Meters of the even relatively young Embedded Systems Metrology domain.

2. SYM²

The advent of different Smart Electricity Meter types, each with proprietary functionality and interfaces, led to a heterogeneous meter park for the Measuring Point Operators. Considerable expenditures for administration and maintenance made heterogeneous parks cost intensive. This effect was recognised by the different stakeholders of the Electricity domain, clearing the way for a common standard. After some consolidations of domain partners, the *tLZ* project group was founded. *tLZ* is an abbreviation for the German equivalent of clock synchronous Load Profile Meter (*t*aktsynchroner *L*astgangs*Z*ähler).
The first version of the specification for the *Synchronous Modular Meter (SYM²)* was issued in 2007. The actual version of the SYM² general specification this paper will deal with is v1.03 (6th of October 2009)[12].
The goal of the SYM² approach and the main characteristics of the SYM² general specification [12] are as follows:

> "The general specification for Synchronous Modular Meters (SYM²) serves to provide development engineers at the meter manufacturers and the staff dealing with invoicing metering equipment at network operators, metering point operators and vendors with a harmonised working document for load profile meters featuring a clocked registration period.
> The goals of the SYM² have been defined as follows under the paramount consideration of cost reduction in operations management:
>
> - Modularised device concept with mounted or integrated function units and a standardised interface, and thus installation of only those elements specifically required on site plus continual easy adaptation to new developments.
> - use of meter readings in place of increments/ power mean values,
> - dispensing with the maximum metering mechanism,
> - dispensing with integrated tariff control,
> - reducing the influence of the device clock by changing to a seconds index,
> - separation into a compulsory-calibration register and additional modules not subject to compulsory calibration for upgrading transparency by returning to simple measured values,
> - reducing the complexity of the individual modules,
> - firmware download utilising the concepts of the WELMEC guideline." [13]

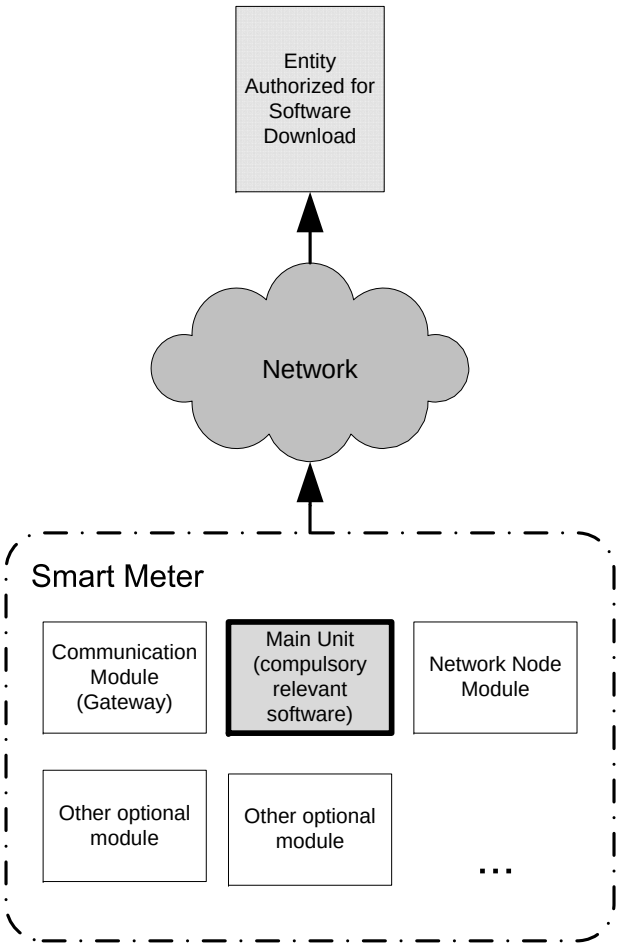

Figure 1: Smart Electricity Meter modular design as proposed in [12]

Modular approaches like SYM^2 feature a structure shown in figure 1. The Smart Meter consists of a main module with sensors for measuring the power consumption and logging the readout values. Due to lawful regulations like the Measuring Instruments Directive (MID 2004/22/EC) [3] from the European Council, the software running the main module directly involved in the measurement aggregation, is compulsory relevant. This means the hardware module itself, as well as the software part has to pass a type approval before being placed on the market. Additional documents concerning lawful regulations and guidelines for their realisation in Smart Meters are the PTB-A 50.7 [10], the WELMEC 7.2 Software Guide [13], [14] and the OIML D31 [9].
For receiving a type approval, the MID offers different assessment procedures. Notified Approval Bodies in the member states of the European Union have the task to accomplish these procedures. When providing a firmware update for the main unit of existing meters in the field, the

meter manufacturer has to forward it to the Approval Body for type approval. Only approved software is allowed to be flashed to the meter. Beside the main unit, a SYM^2 Meter can also contain internal modules like a Communication module, a Pulse Emitting module or a Network Node module. Other customer specific modules are possible too, underlining the modular approach of SYM^2.

2.1 Software Download Concepts

Both variants of modules mentioned in the chapter before, the ones subject to compulsory calibration (main unit), as well as other optional modules (Communication module, Pulse Emitting module, Network Node module, etc.) can be updated in the field by using Software Download mechanisms. In the following they are explained using sequence charts.

2.1.1 Software Download (compulsory calibration relevant)

Figure 2 taken from the SYM^2 general specification shows the concept of the Software Download for the main module (compulsory calibration relevant software) of the meter.
The main characteristic is the use of asymmetric cryptography for digital signatures and their verification. Basic requirement is a A public/private key pair from the Approval Body. The Approval Body uses its private key for signing

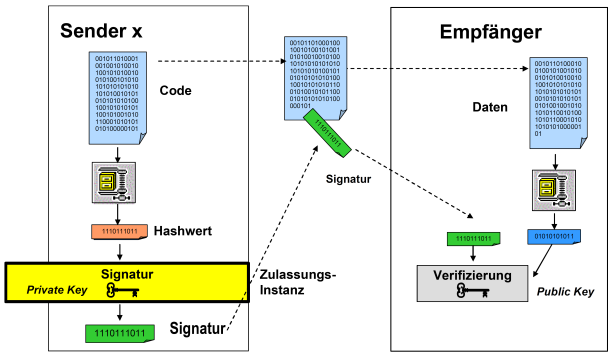

Figure 2: SYM^2 Software Download concept (compulsory calibration relevant software) from [12]

the new software image for the meter. The meter itself is in permanent possession of the corresponding public key from the Approval Body, enabling the verification of signatures for new firmwares.
It is of great importance, that the public key is deposited in a secure storage of the meter, making any kind of manipulation impossible. The exact chronological order of the procedure is depicted in figure 3. The sequence chart shows action lines for three roles, the Approval Body, the Measuring Point Operator and the Smart Meter. The first action of a complete Software Download sequence is the provision of a new software image by the Measuring Point Operator. The image is then forwarded to the Approval Body and runs through the type approval process.

In case of success, the Approval Body calculates a hash over the software image, signs it, and returns it to the Measuring Point Operator. After receiving the Approval Body's signature, the Measuring Point Operator transmits it to the Smart Meter, together with the software image.
The Smart Meter receives image and signature. Afterwards, the signature is verified with the public key of the Approval Body, which is stored in the secure memory of the Smart Meter. The new software image is solely accepted if the signature can be successfully verified. For starting the new firmware, the Smart Meter initiates the local software update routine. The final step comprises the action of the Smart meter to boot the new firmware. The security mechanism used to assure the correct origin of the software image is a digital signature using asymmetric cryptography.

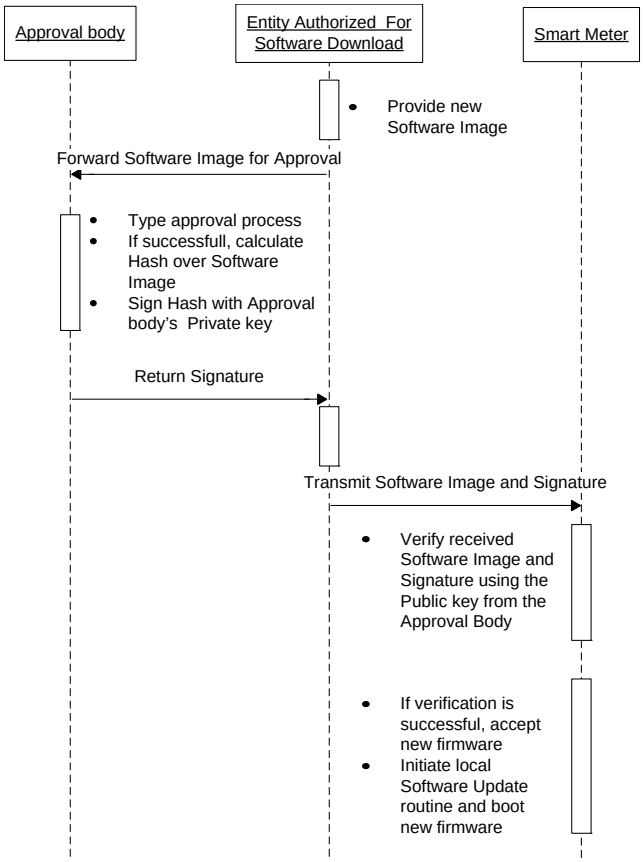

Figure 3: Sequence diagram of SYM^2 Software Download mechanism (compulsory calibration relevant software)

2.1.2 Software Download (Non-compulsory calibration relevant)

For optional modules with non-compulsory relevant software, the SYM^2 general specification proposes a second variant of software download. Figure 4 shows the general mechanism. Since the software is non-compulsory relevant, it doesn't have to be approved by an Approval Body.
The requirement for the correct origin of the new firmware exists in the same way as for the compulsory relevant parts. The Measuring Point Operator has to be sure only his software images can be flashed to the Smart Meter. This is accomplished by using secrets, only known to the Smart Meter manufacturer or Measuring Point Operator and the Smart

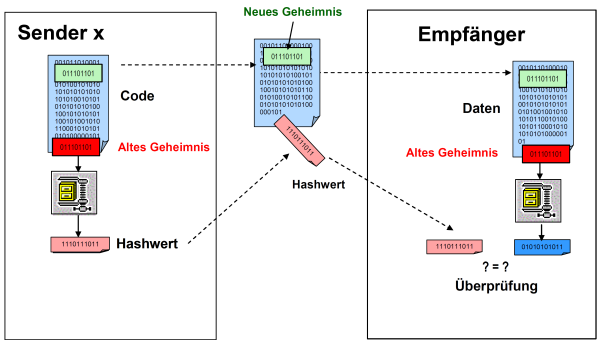

Figure 4: SYM^2 Software Download concept (non-compulsory calibration relevant software) from [12]

Meter itself. Precondition for the software update sequence shown in figure 5 is an already existing secret (between the Measuring Point Operator and the Smart Meter) in a secure storage of the Smart Meter, which cannot be manipulated. Of course, this secret must be known to the Measuring Point Operator. Every time he provides a new software image, a new secret is generated and embedded into the image. In the next step, the software image containing the new secret is enhanced by extending the old secret (the same as in the secure storage of the meter) and a hash is calculated.
Hash and software image with embedded new secret are transmitted to the Smart Meter in the field. After successfully receiving the image and hash, the Meter appends the old secret to the image. The correct hash value is verified by recalculating it with identical parameters. If both hash values are equal, the new firmware is accepted and the old secret in the Meter is replaced by the new one from the image.
Afterwards, the Meter initialises the local software update routine and boots the new firmware.

The technique of using a shared secret to ensure authenticity and integrity of the firmware image is known as Message Authentication Code (MAC) and will be referred to during the formal investigation in section 4.

3. THE SECURITY MODELLING FRAMEWORK SEMF

In this section we briefly introduce the Security Modeling Framework SeMF and, based on this, the formal definition of the properties and implications amongst them to be used in the investigation of the SYM^2 specification.

SeMF is a framework based on formal language theory that provides a formal semantics for the specification of models of systems, for the definition of security properties in terms of these models, and for the reasoning about security properties holding or not holding in certain system models. In contrast to e.g. Temporal Logics [11] that aims at more general properties, SeMF specifically targets security properties. Work on SeMF started in the early 2000s with [6, 7, 8] and the framework is continuously being extended. As opposed to e.g. BAN [1] properties are defined based on the underlying formal language framework to allow in depth analysis of implications amongst them.

3.1 Basis of SeMF

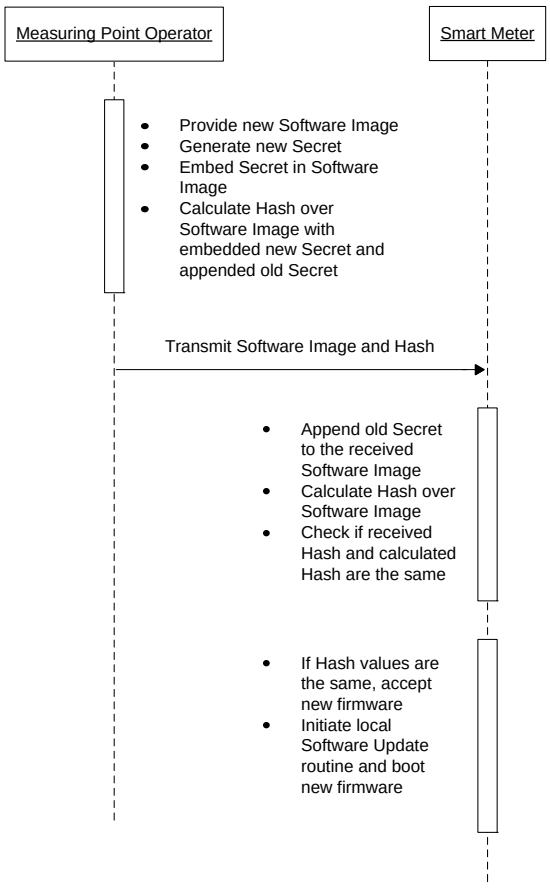

Figure 5: Sequence diagram of SYM^2 Software Download mechanism (non-compulsory calibration relevant software)

The behaviour B of a discrete system S can be formally described by the set of its possible sequences of actions (traces). Therefore $B \subseteq \Sigma^*$ holds, where Σ (called the alphabet) is the set of all actions of the system, Σ^* is the set of all finite sequences (called words) of elements of Σ, including the empty sequence denoted by ε. Subsets of Σ^* are called formal languages. Words can be composed: if u and v are words, then uv is also a word. For a word $x \in \Sigma^*$, we denote the set of actions of x by $alph(x)$. For more details on the theory of formal languages we refer the reader to [2].

We further extend the system specification by two components: *agents' initial knowledges* about the global system behaviour and *agents' local views*. The initial knowledge $W_P \subseteq \Sigma^*$ of agent P about the system consists of all traces P initially considers possible, i.e. all traces that do not violate any of P's assumptions about the system. Every trace that is not explicitly forbidden can happen in the system. An agent P may assume for example that a message that was received must have been sent before. Thus the agent's W_P will contain only those sequences of actions in which a message is first sent and then received. Further we can

assume $B \subseteq W_P$, as reasoning within SeMF primarily targets the validation and verification of security properties in terms of positive formulations, i.e. assurances the agents of the system may have. Other approaches that deal with malfunction, misassumptions and attacker models cannot rely on this assumption.

In a running system, P can learn from actions that have occurred. Satisfaction of security properties obviously also depends on what agents are able to learn. After a sequence of actions $\omega \in B$ has happened, every agent P can use its *local view* λ_P – an alphabetic language homomorphism $\lambda_P : \Sigma^* \to \Sigma_P^*$ – of ω to determine the sequences of actions it considers to have possibly happened. Examples of an agent's local view are that an agent can see only its own actions, or that an agent P can see that an action $send(sender, message)$ occurred but cannot see the message, in which case $\lambda_P(send(sender, message)) = send(sender)$, or an agent may see a message on a network bus and is not able to determine the sender $\lambda_P(send(sender, message)) = send(message)$.

For a sequence of actions $\omega \in B$ and agent $P \in \mathbb{P}$ ($\mathbb{P}$ denoting the set of all agents), $\lambda_P^{-1}(\lambda_P(\omega)) \subseteq \Sigma^*$ is the set of all sequences that look exactly the same from P's local view after ω has happened. Depending on its knowledge about the system S, including underlying security mechanisms and system assumptions, P does not consider all sequences in $\lambda_P^{-1}(\lambda_P(\omega))$ possible. Thus it can use its initial knowledge to reduce the set: $\lambda_P^{-1}(\lambda_P(\omega)) \cap W_P$, which describes all sequences of actions P considers to have possibly happened when ω has happened.

In order to model these boot cycles, we use the definition of a *phase* provided in [5]. A phase $V \subset \Sigma^*$ is a prefix closed language consisting only of words which, as long as they are not maximal in V, show the same continuation behavior within V as within B.

DEFINITION 1. *Let $B \subseteq \Sigma^*$ be a system. A prefix closed language $V \subset \Sigma^*$ is a phase in B if the following holds:*

1. $V \cap \Sigma \neq \emptyset$
2. $\forall \omega \in B$ *with* $\omega = uv$ *and* $v \in V \setminus (max(V) \cup \{\varepsilon\})$ *holds:* $\omega^{-1}(B) \cap \Sigma = v^{-1}(V) \cap \Sigma$

Thus a phase as defined above is essentially a part of the system behaviour that is closed with respect to concatenation. In analogy to the maximal words of a phase V which are those $v \in V$ for which exists $\omega, u \in B$ with $\omega = uv$ such that for all $a \in \Sigma$ with $\omega a \in B$ holds $va \notin V$, we define the minimal words of a phase as all $v \in V$ with $|alph(v)| = 1$.

A phase can be a very complex construct. However, in many cases phases are of interest that can be defined by their starting and ending actions. Since an action can occur more than once in a word, it is not sufficient to identify the starting and terminating actions for determining where a particular phase starts and where it ends. The following definition takes this into account.

In the following, we denote the number of occurrences of a set of actions $\Gamma \in \Sigma$ in a word ω by $card(\Gamma, \omega)$. If Γ consists of only one action a, we simply say $card(a, \omega)$.

DEFINITION 2. *Let $s_1, \ldots, s_k, t_1, \ldots, t_l \in \Sigma$ be actions. Then $V(\{s_1, \ldots, s_k\}, \{t_1(j_1), \ldots, t_l(j_l)\})$ (with $j_1, \ldots, j_l \in \mathbb{N}$) defines a phase in B that starts with actions $s_1, \ldots, s_k$ and terminates with actions $t_1, \ldots, t_l$ in the following sense:*

- *For all $\omega \in B$ for which exists $s_j \in \{s_1, \ldots, s_k\}$ such that $\omega s_j \in B$ follows $s_j \in V(\{s_1, \ldots, s_k\}, \{t_1(j_1), \ldots, t_l(j_l)\})$ (i.e. s_j is minimal in $V(\{s_1, \ldots, s_k\}, \{t_1(j_1), \ldots, t_l(j_l)\})$).*
- *For all $\omega \in B$ for which exists $t_i(j_i) \in \{t_1(j_1), \ldots, t_l(j_l)\}$ and $u, v \in \Sigma^*$ with $\omega = uvt_i$, $vt_i \in V(\{s_1, \ldots, s_k\}, \{t_1(j_1), \ldots, t_l(j_l)\})$, and $card(t_i, vt_i) = j_i$ follows that vt_i is maximal in $V(\{s_1, \ldots, s_k\}, \{t_1(j_1), \ldots, t_l(j_l)\})$.*

The $\emptyset$ as start action will thereby denote those phases starting at the beginning of the system lifetime, whilst $\emptyset$ as end action will denote phases not stopping once startet.
In the above definition, the starting action(s) need to be fixed so that starting from these the terminating actions occurring in the phase can be counted in order to identify those ones that actually terminate the phase. Note that in the cases where no cardinality is assigned to terminating actions, we assume a cardinality of 1.

3.2 Properties

Security properties can now be defined in terms of the agents' initial knowledges and local views. In [8] a variety of definitions of security properties (e.g. authenticity, proof of authenticity, confidentiality) is introduced. For this paper only a subset of these properties, that do not respect agent assurance but only investigate functional correctness, will be used.

In order to express that a certain action never happens within the complete system lifetime or only within a phase, we use the following properties:

DEFINITION 3. *For a system with behaviour $B \subseteq \Sigma^*$, $A \subseteq \Sigma$, $nothappens(A)$ holds if for all $\omega \in B$: $A \cap alph(\omega) = \emptyset$.*

DEFINITION 4. *For a system with behaviour $B \subseteq \Sigma^*$ and a phase V in B, $a \subseteq \Sigma$, $nothappens\text{-}wi\text{-}phase(A, V)$ holds if for all $v \in V$: $A \cap alph(v) = \emptyset$.*

Further we need to express, that whenever an action out of the set B has happened, an action out of the set A must have happened (before):

DEFINITION 5. *For a system with behaviour $B \subseteq \Sigma^*$, $A, B \subseteq \Sigma$, $precede(A, B)$ holds if for all $\omega \in B$, if $B \cap alph(\omega) \neq \emptyset$ then $a \cap alph(\omega) \neq \emptyset$.*
Note that since the property needs to hold for all ω in B, it holds in particular for an ω with an action from B as the last action. Hence an action from A must happen *before* this action from B.

And finally we provide a specialization of this property to further confine the point in the system lifetime when such an action from A shall have happened:

DEFINITION 6. *For a system with behavior $B, V \subseteq \Sigma^*$, $a, b \subseteq \Sigma$, $precedephase(a, b, V)$ holds if $\forall \omega \in B$ with $b \cap alph(\omega) \neq \emptyset$, there exists $u, v, w \in \Sigma^*$ such that $\omega = uvw, v \in V$ and $a \cap alph(v) \neq \emptyset$.*

3.3 Implications

We will now provide a set of Theorems and Lemmata that represent implications to be used in the later proof attempts, starting with an obvious relation between *nothappens* properties:

LEMMA 1. *Given a system S and actions $A, B \subseteq \Sigma$ and phase V, if $nothappens(A)$ and $nothappens(B)$ hold in S, then $nothappens(A \cup B)$ holds in S.*

Next we investigate the transitivity of *precede* and *precedephase* respectively.

THEOREM 1. *Given a system S and actions $a, b, c \in \Sigma$, if*

precede(a, b) *and* *precede*(b, c) *hold in* S*, then* *precede*(a, c) *holds in* S *as well.*

PROOF 1. *precede*(b, c) *states that for all* $\omega \in B$ *with* $c \in alph(\omega)$ *it holds that* $b \in alph(\omega)$*.* *precede*(a, b) *states for exactly those* ω *that* $a \in alph(\omega)$*, which can be expressed as* *precede*(a, c)*.*

LEMMA 2. *Given a system* S *and actions* $a, b, c \in \Sigma$*, if* *precedephase*(a, b) *and* *precede*(b, c) *hold in* S*, then* *precedephase*(a, c) *holds in* S *as well.*

Finally, we investigate the relations between *precede* and *nothappens*

THEOREM 2. *Given a system* S *and actions* $a, b \in \Sigma$*, if* *nothappens*(b) *holds in* S*, then* *precede*(a, b) *holds in* S*.*

PROOF 2. *Assuming* *precede*(a, b) *would not hold in* S*, then there must be* $\omega \in B$ *with* $b \in alph(\omega)$ *and* $a \notin alph(\omega)$*.* $b \in alph(\omega)$ *however is a contradiction to* *nothappens*(b)*.*

THEOREM 3. *Given a system* S *and actions* $a, b, c \subseteq \Sigma$*, if* *precede*(a, b) *and* *nothappens*(c) *holds in* S*, then* *precede*$(a, b \cup c)$ *holds in* S*.*

PROOF 3. *precede*$(a, b \cup c)$ *states that* $\forall \omega \in B$ *with* $alph(\omega) \cap (b \cup c) \neq \emptyset$ *that also* $alph(\omega) \cap a \neq \emptyset$*. For the case of* $alph(\omega) \cap b \neq \emptyset$ *we know from* *precede*(a, b) *that* $alph(\omega) \cap a \neq \emptyset$*. For the case of* $alph(\omega) \cap c \neq \emptyset$ *we know that this case never happens due to* *nothappens*(c)*.*

THEOREM 4. *Given a system* S *and actions* $a, b \in \Sigma$*, if* *precede*(a, b) *and* *nothappens*(a) *hold in* S*, then* *nothappens*(b) *holds in* S*.*

PROOF 4. *Assuming* *nothappens*(b) *would not hold in* S*, then there must be* $\omega \in B$ *with* $b \in alph(\omega)$*.* *precede*(a, b) *would further require, that* $a \in alph(\omega)$*, which would be a contradiction to* *nothappens*(a)*.*

LEMMA 3. *Given a system* S*, actions* $a, b \in \Sigma$ *and a phase* V*, if* *precede*(a, b) *and* *nothappens-wi-phase*(a, V) *hold in* S*, then* *nothappens-wi-phase*(b, V) *holds in* S*.*

THEOREM 5. *Given a system* S *and action* $a \in \Sigma$*, if* *nothappens*(a) *holds in* S *then for all phases* V *in* S *nothappens-wi-phase(a,V) holds as well*

PROOF 5. *Assuming* *nothappens*(a, V) *would not hold in* S*, then there must be* $v \in V$ *with* $a \in alph(v)$*. Therefore according to the definition of phases, there must be* $\omega \in B$ *such that* $a \in alph(\omega)$ *which is a contradiction to* *nothappens*(b)*.*

4. FORMAL INVESTIGATION

The formalisms given in Section 3 serve as basis for the investigation of important security requirements towards a Secure Software Download. For both variants, the correct origin of the software image, as well as the freshness of this image will be investigated. Note that this is *only* an investigation and not a validation as the latter would require significantly more effort.

4.1 Software Download (Compulsory calibration relevant)

The first investigation approach demonstrates the the correct origin of the software image downloaded to the Smart Meter.

4.1.1 Investigating Correct Origin

Basis for the formal investigation is the SYM^2 Secure Software Download Sequence for the compulsory relevant software (see Figure 2) and the actions denoted in the sequence chart (see Figure 3).

System Definition.

The Software Download variant for compulsory relevant parts of the meter software involves the following agents and their abbreviations:

- *Approval Body (AB)*
  The Approval Body is the national Notified Body in charge of the compulsory calibration. Basis for the calibration is the MID [3] with its different assessment modules. Only software approved by this body is allowed to be downloaded to the meter.
- *Measuring Point Operator (MPO)*
  The Measuring Point Operator is the owner of the Smart Meter. He is in charge of administrating and maintaining the device in the field. He is in the position to download software to the meter.
- *Smart Meter (SM)*
  The Smart Meter is the device in the field being supplied with a new software image. Download of this software image is initiated by the Measuring Point Operator using the Secure Software Download sequence.

$$\mathbb{P} = \{AB, MPO, SM\}$$

These agents are able to perform the following set of actions:

- $provide(MPO, fw_i)$
  Measuring Point Operator MPO provides a new firmware fw_i to the AB for approval.
- $approve(AB, fw_i)$
  Approval Body AB performs a type approval for firmware fw_i.
- $sign(AB, fw_i, PrivK_j, sig_j(fw_i))$
  Approval Body AB signs a firmware fw_i with its private key $PrivK_j$ and returns signature $sig_j(FW_i)$ of the firmware fw_i. Note that the relation between the signature and the utilised private key is denoted through their indexes.
- $transmit(MPO, fw_i, sig_j(fw_k))$
  Measuring Point Operator MPO transmits a new firmware fw_i and signature sig_j to the Smart Meter.
- $deploy(MPO, SM(fw_i))$
  Measuring Point Operator MPO deploys a Smart Meter SM with a firmware fw_i.
- $verify(SM(fw_i), fw_j, sig_k(fw_j), PubK_k)$
  Smart Meter SM running firmware fw_i verifies a newly received firmware fw_j with signature $sig_k(fw_j)$ using a public key $PubK_k$. Note that the relation between the signature and the utilised public key is denoted through their indexes. Only equal indexes allow for this action to occur.
- $flash(SM(fw_i), fw_j)$
  Smart Meter SM running firmware fw_i flashes the new firmware fw_j to the memory.
- $boot(SM, fw_i)\}$
  Smart Meter SM boots firmware fw_i.

$$\Sigma = \{provide(MPO, fw_i),$$
$$approve(AB, fw),$$
$$sign(AB, fw_i, PrivK_j, sig_j(fw_i)),$$
$$transmit(MPO, fw_i, sig_j(fw_k)),$$
$$deploy(MPO, fw_i, sig_j(fw_k)),$$
$$verify(SM(fw_i), fw_j, sig_k(fw_j), PubK_k),$$
$$flash(SM(fw_i), fw_j),$$
$$boot(SM, fw_i)\}$$

Requirement.

It is stated in the SYM^2 specification, that only approved firmwares are allowed to be flashed to the Smart Meter. We will interpret this as: Every firmware $Firmware(FW_i)$ flashed on a smart meter must have been approved by the Approval body:

$$precede(approve(AB, fw_i), flash(SM(fw_j), fw_i)) \quad \text{(Req1)}$$

Assumptions.

The following set of assumptions will be used for the proof:

1. The properties of elliptic curve cryptography (ECC) used in the metrology domain, ensure that a signature sig_j verified with a public key $PubK_j$ must have been generated with the corresponding private key $PrivK_j$. (Note that we disregard the investigation of the $PrivK$'s confidentiality here)

$$precede(sign(AB, fw_i, PrivK_j, sig_j(fw_i)),$$
$$verify(SM(fw_k), fw_i, sig_j(fw_i), PubK_j))$$

2. Every firmware is approved before it is signed by the Approval Body:

$$precede(approve(AB, fw_i),$$
$$sign(AB, fw_i, PrivK_{AB}, sig_{AB}(fw_i)))$$

3. The Measuring Point Operator will only deploy Smart Meters to the customers' sites with approved firmwares:

$$precede(approve(AB, fw_i), deploy(MPO, SM(fw_i)))$$

4. The Smart Meter can only boot the firmware that it was deployed with or that has been flashed:

$$precede(\{flash(SM(fw_i), fw_j),$$
$$deploy(MPO, SM(fw_j))\},$$
$$boot(SM, fw_j))$$

5. A Smart Meter running an approved firmware will verify a newly received firmware's signature with the Approval Body's public key before flashing it.

$$\forall fw_k \in appr\text{-}fws :$$
$$precede(\{verify(SM(fw_k), fw_i, sig_j(fw_i), PubK_j)$$
$$\mid j = AB\},$$
$$flash(SM(fw_k), fw_i))$$

6. The execution of an action by the Smart Meter under a given firmware fw_i requires the Smart Meter to have booted this firmware:

$$precede(boot(SM, fw_i), \Sigma_{/SM(fw_i)})$$

Proof.

We define that any firmware that is approved by the Approval Body belongs to the set of approved firmwares:

$$nothappen(\{approve(AB, fw_i)$$
$$\mid fw_i \notin appr\text{-}fws\}) \quad (1)$$

From Assumptions 2. and 1. we may conclude with Theorem 1 that whenever a verification of a firmware's signature with the public key of the Approval Body succeeds, this firmware has been approved by the Approval Body.

$$precede(approve(AB, fw_i),$$
$$verify(SM(fw_k), fw_i, sig_{AB}(fw_i), PubK_{AB})) \quad (2)$$

Therefore using the Proof Step (1) any firmware successfully verified with the Approval Body's Public Key is an approved Firmware through Theorem 4:

$$nothappen(\{verify(SM(fw_k), fw_i, sig_{AB}(fw_i), PubK_{AB})$$
$$\mid fw_i \notin appr\text{-}fws\}) \quad (3)$$

Through Assumption 5. we may further conclude using Theorem 4 that an approved firmware will only flash firmwares that are approved as well:

$$\forall fw_k \in appr\text{-}fws :$$
$$nothappen(\{flash(SM(fw_k), fw_i)$$
$$\mid fw_i \notin appr\text{-}fws\}) \quad (4)$$

Proof Statement (1) and Assumption 3. imply through Theorem 4 that only approved firmwares are deployed by the MPO:

$$nothappen(\{deploy(MPO, SM(fw_i)) \mid fw_i \notin appr\text{-}fws\}) \quad (5)$$

This is of course especially true for the phase from system start up to the first boot of the Smart Meter by Theorem 5:

$$nothappen\text{-}wi\text{-}phase(\{deploy(MPO, SM(fw_i))$$
$$\mid fw_i \notin appr\text{-}fws\},$$
$$V(\emptyset, boot(SM, fw_j)(1))) \quad (6)$$

As $boot(SM, fw_j)$ is the last action within this phase, we may trivially conclude from Assumption 6. that no other actions $\Sigma_{/SM(fw_i)}$ of the Smart Meter happen within this phase:

$$nothappen\text{-}wi\text{-}phase(\Sigma_{/SM(fw_i)} \setminus \{boot(SM, fw_j)\},$$
$$V(\emptyset, boot(SM, fw_j)(1))) \quad (7)$$

This is especially true for $flash(SM(fw_i), fw_k) \in \Sigma_{/SM(fw_i)} \setminus \{boot(SM, fw_j)\}$.

This statement together with Assumption 4. let us conclude throug Lemma 3 that the first boot is performed with an approved firmware:

$$nothappen\text{-}wi\text{-}phase(\{boot(SM, fw_i) \mid fw_i \notin appr\text{-}fws\},$$
$$V(\emptyset, boot(SM, fw_j)(1))) \quad (8)$$

We will use this as the anchor for an inductive proof over the number of boot cycles of the smart meter.

As induction step, we start by assume that up to and including the n-th boot cycle, only approved firmwares have been booted.

$$\begin{aligned} &\textit{nothappen-wi-phase}(\{\textit{boot}(SM, fw_i) \mid fw_i \notin \textit{appr-fws}\}, \\ &\qquad V(\emptyset, \textit{boot}(SM, fw_j)(n))) \end{aligned} \tag{9}$$

We know from Assumption 6. through Lemma 3 that only approved firmwares' actions will performed up to the $n+1$-th boot cycle

$$\begin{aligned} &\textit{nothappen-wi-phase}(\{a \in \Sigma_{/SM(fw_i)} \mid fw_i \notin \textit{appr-fws}\} \\ &\qquad V(\emptyset, \textit{boot}(SM, fw_j)(n+1))) \end{aligned} \tag{10}$$

which is especially true for $\{flash(SM(fw_k), fw_i) \mid fw_k \notin \textit{appr-fws}\} \subseteq \{a \in \Sigma_{/SM(fw_i)} \mid fw_i \notin \textit{appr-fws}\}$.

According to Proof Statement (4) this means by Theorem 5 and Lemma 1 that only approved firmwares were flashed:

$$\begin{aligned} &\textit{nothappen-wi-phase}(\{flash(SM(fw_k), fw_i) \\ &\qquad \mid fw_i \notin \textit{appr-fws}\}, \\ &\qquad V(\emptyset, \textit{boot}(SM, fw_j)(n+1))) \end{aligned} \tag{11}$$

By Assumption 4. and Lemma 3 this means that also the $n+1$-th boot cycle is performed with an approved firmware:

$$\begin{aligned} &\textit{nothappen-wi-phase}(\{\textit{boot}(SM, fw_i) \mid fw_i \notin \textit{appr-fws}\}, \\ &\qquad V(\emptyset, \textit{boot}(SM, fw_j)(n+1))) \end{aligned} \tag{12}$$

Taking Proof Statement (8) as an anchor and the conclusion from Proof Statement (9) to (12) as a step we can perform an inductive proof over the number of boot cycles of the smart meter to conclude that only approved firmwares are booted by the Smart Meter:

$$\textit{nothappen}(\{\textit{boot}(SM, fw_i) \mid fw_i \notin \textit{appr-fws}\}) \tag{13}$$

With this information it is easy to use Assumption 6. and Theorem 4 to conclude

$$\textit{nothappen}(\{a \in \Sigma_{/SM(fw_i)} \mid fw_i \notin \textit{appr-fws}\}) \tag{14}$$

and then to generalize Assumption 5. and conclude using Proof Statement (2) that only approved firmware is flashed on the Smart Meter:

$$\textit{precede}(\textit{approve}(AB, fw_i), flash(SM(fw_j), fw_i)) \tag{15}$$

$q.e.d.$

Analysis.

The above proof of a certain aspect of the software download yields the following possibility for analysis:

- It can be seen that given the presented assumptions, the requirement can be fulfilled.
- However especially Assumption 5. will be difficult to fulfil in practice. It describes a certain behavior of approved firmwares that may be prone to errors through exploitable bugs in practical applications.
- This is especially important as the proof requires the inductive iteration through all boot cycles of the Smart Meter and all firmwares it has been booted with in the past. This may be circumvented by establishing e.g. hardware based concepts for secure and/or authenticated boot following Trusted Computing concepts.
- Further it is not assured that a given firmware suits a given Smart Meter. Cross-flashing of actually incompatible firmwares and Smart Meters may further harden the aforementioned issues. A possible countermeasure could be to include type information into the firmware and check this as well, or to use a given $PubK_{AB}$ for a single type of Smart Meters only.
- Finally, downgrading and upgrade cycling attacks of firmwares are possible. This will be further investigated in the following section.

4.1.2 Investigating Correct and Fresh Origin

As mentioned during the analysis in the previous section we will now further investigate the freshness requirements of the firmware download. We will use the same system and assumptions as defined in Section 4.1.1.

Requirement.

The new requirement to be investigated is the correct and fresh origin of the flashed firmware. *Fresh* here means that the firmware was issued *after* the firmware currently running on the Smart Meter:

$$\begin{aligned} &\textit{precedephase}(\textit{approve}(AB, fw_i), \\ &\qquad flash(SM(fw_j), fw_i), \\ &\qquad V(\textit{approve}(AB, fw_j), \emptyset)) \end{aligned} \tag{Req2}$$

Proof.

The proof can be performed the same as in Section 4.1.1. However this time, the resulting property

$$\textit{precede}(\textit{approve}(AB, fw_i), flash(SM(fw_j), fw_i)) \tag{15}$$

does not meet the requirement. Further there are no means or assumptions to further constraint the phase in which the $\textit{approve}(AB, fw_i)$ action is performed. $fail$

Analysis.

This failed proof attempt demonstrates the following misuse cases:

- An attacker could "downgrade" a Smart Meter to an older version. This cannot be detected by the Smart Meter.
- An attacker could "boot cycle" a Smart Meter with the same version. As the commands to download a new firmware and activate (boot) it are "Level 0" commands, they are not protected by any further means than the firmware's signature. Assuming a Smart Meter will only reboot, when it flashed a new firmware, an attacker could command the Smart Meter to download the same firmware over and over and boot it each time.
- This attack could be used to undermine the availability of a Smart Meter, possibly obfuscating client consumption or confusing the smart grid.
- This attack could also be used to fill the flash log, triggering possible overrun attacks there or obfuscating other attacks, such as the downgrading of flashing of an unapproved firmware.
- These attacks could be circumvented by adding a version number to the firmware and checking it against the current firmwares version during flash. This coun-

termeasure will be further investigated in the following section.

4.1.3 Improved Correct and Fresh Origin

In order to circumvent the aforementioned attacks of downgrading and boot cycling we will improve the specification by adding a version number to the firmware.

Extended System.

The new system still consists of the same agents as in Section 4.1.1:

$$\mathbb{P} = \{AB, MPO, SM\}$$

The agents' action are also the same, however each occurrence of fw has been extended by a version number:

$$\begin{aligned}\Sigma = \{&provide(MPO, fw_i(v_i)),\\ &approve(AB, fw_i(v_i)),\\ &sign(AB, fw_i(v_i), PrivK_j, sig_j(fw_i(v_n))),\\ &transmit(MPO, fw_i(v_i), sig_j(fw_k(v_k))),\\ &deploy(MPO, fw_i(v_i), sig_j(fw_k(v_k))),\\ &verify(SM(fw_i(v_i)), fw_j(v_j), sig_k(fw_j(v_j)), PubK_k),\\ &flash(SM(fw_i(v_i)), fw_j(v_j)),\\ &boot(SM, fw_i(v_i))\}\end{aligned}$$

Requirement.

The requirement to be investigated is the correct and fresh origin of the flashed firmware. *Fresh* here means that the firmware was issued *after* the firmware currently running on the smart meter:

$$\begin{aligned}precedephase(&approve(AB, fw_i),\\ &flash(SM(fw_j), fw_i),\\ &V(approve(AB, fw_j), \emptyset))\end{aligned} \quad \text{(Req2)}$$

Extended Assumptions.

We will utilise all the assumptions from Section 4.1.1, but extend them by the following additional assumptions:

7. Any approved firmware will only flash firmwares that have a version number higher than their own:

$$\begin{aligned}&\forall fw_i \in appr\text{-}fws :\\ &nothappen(\{flash(SM(fw_j(v_j)), fw_i(v_i)) \mid v_j >= v_i\})\end{aligned}$$

8. The Approval Body will only issue firmwares with version information in ascending order, expressed as:

$$\begin{aligned}nothappen\text{-}wi\text{-}phase(&\{approve(AB, fw_i(v_i))\\ &\mid v_i > v_j\},\\ &V(\emptyset, approve(AB, fw_j(v_j))))\end{aligned}$$

Proof.

This proof includes all the proof steps from Section 4.1.1's proof up to Proof Statement (15) expressed for the new set of actions:

$$\begin{aligned}precede(&approve(AB, fw_i(v_i)),\\ &flash(SM(fw_j(v_j)), fw_i(v_i)))\end{aligned} \quad (15)$$

This is especially true for those $flash$ actions, where $v_i > v_j$. Note that we split the index i into i and k for now.

$$\begin{aligned}precede(&\{approve(AB, fw_i(v_i)) \mid v_i = v_k > v_j\},\\ &\{flash(SM(fw_j(v_j)), fw_k(v_k))) \mid v_k > v_j\})\end{aligned} \quad (16)$$

As we know from Assumption 7. that a $flash$-action will not happen with the first firmware version bigger than the second anyways. Therefore we can conclude with Theorem 3:

$$\begin{aligned}precede(&\{approve(AB, fw_i(v_i)) \mid v_i = v_k > v_j\},\\ &flash(SM(fw_j(v_j)), fw_k(v_k))))\end{aligned} \quad (17)$$

We know from Assumption 8. that a firmware with version v_i will not be approved in the phase from system start until $approve(AB, fw_l(v_l))$ with a version $v_l < v_i$. Accordingly, if an *approve* with a version v_i is known of have preceded, it must have been within the phase after the *approve* with v_l. We may therefore conclude

$$\begin{aligned}&\forall v_i, v_l \text{ with } v_i > v_l :\\ &precedephase(\{approve(AB, fw_i(v_i))\\ &\qquad \mid v_i = v_k > v_j\},\\ &\qquad flash(SM(fw_j(v_j)), fw_k(v_k))),\\ &\qquad V(approve(AB, fw_l(v_l)), \emptyset))\end{aligned} \quad (18)$$

This property especially includes the case of $l = j$ and leads to

$$\begin{aligned}precedephase(&\{approve(AB, fw_i(v_i))\\ &\mid v_i = v_k > v_j\},\\ &flash(SM(fw_j(v_j)), fw_k(v_k))),\\ &V(approve(AB, fw_j(v_j)), \emptyset))\end{aligned} \quad (19)$$

We may now unite the indices $k = i$ again.

$$\begin{aligned}precedephase(&approve(AB, fw_i(v_i))\\ &flash(SM(fw_j(v_j)), fw_i(v_i))),\\ &V(approve(AB, fw_j(v_j)), \emptyset))\end{aligned} \quad (20)$$

which proofs the requirement. *q.e.d.*

Analysis.

It could be shown that the improvement of adding a version number check assures freshness of the firmware. However the other issues such as the reliance on an induction over all previously booted firmwares remain.

4.2 Software Download (Non-compulsory calibration relevant)

We will now investigate the non-relevant software download as described in Section 2.1.

4.2.1 Investigating Correct and Fresh Origin

We will directly investigate the stronger property of correct and fresh origin of firmwares.

System.

The Software Download variant for non compulsory relevant parts of the meter software involves the following agents and their abbreviations:

- *Measuring Point Operator (MPO)*

The Measuring Point Operator is the owner of the Smart Meter. He is in charge of administrating and maintaining the device in the field. He is in the position to issue new software and download software to the meter.

- *Smart Meter (SM)*
  The Smart Meter is the device in the field being supplied with a new software image. Download of this software image is initiated by the Measuring Point Operator using the Secure Software Download sequence.

$$\mathbb{P} = \{MPO, SM\}$$

These agents are able to perform the following set of actions:

- $provide(MPO, fw_i(SS_i))$
  The MPO provides (program, compile and package) a new firmware including a new Shared Secret.
- $sign(MPO, fw_i(SS_i), SS_j, mac_j(fw_i(SS_i)))$
  MPO signs a firmware $fw_i(SS_i)$ with its shared secret SS_j and returns message authentication code $mac_j(FW_i(SS_i))$ of the firmware. Note that the relation between the message authentication code and the utilised key is denoted through their indexes.
- $transmit(MPO, fw_i(SS_i), mac_j(fw_k(SS_k)))$
  Measuring Point Operator MPO transmits a new firmware $fw_i(SS_i)$ and message authentication code $mac_j(fw_k(SS_k))$ to the Smart Meter.
- $deploy(MPO, SM(fw_i(SS_i)))$
  Measuring Point Operator MPO deploys a new Smart Meter SM with a firmware firmware fw_i including a shared secret SS_i.
- $verify(SM(fw_i(SS_i)), fw_j(SS_j), mac_i(fw_j(SS_j)))$
  Smart Meter SM running firmware $fw_i(SS_i)$ verifies a newly received firmware fw_j with message authentication code $mac_k(FW_j)$ using its shared secret SS_i. Note that the relation between the message authentication code and the utilised key is denoted through their indexes. Only equal indexes allow for this action to occur.
- $flash(SM(fw_i(SS_i)), fw_j(SS_j))$
  Smart Meter SM running firmware fw_i flashes the new firmware fw_j to the memory.
- $boot(SM, fw_i(SS_i))\}$
  Smart Meter SM boots firmware fw_i.

$$\begin{aligned}\Sigma = \{&provide(MPO, fw_i(SS_i)),\\ &sign(MPO, fw_i(SS_i), SS_j, mac_j(fw_i(SS_i))),\\ &transmit(MPO, fw_i(SS_i), mac_j(fw_k(SS_k))),\\ &deploy(MPO, SM(fw_i(SS_i))),\\ &verify(SM(fw_i(SS_i)), fw_j(SS_j), mac_i(fw_j(SS_j))),\\ &flash(SM(fw_i(SS_i)), fw_j(SS_j)),\\ &boot(SM, fw_i(SS_i))\}\end{aligned}$$

Requirement.

The requirement to be investigated is the correct and fresh origin of the flashed firmware. *Fresh* here means that the firmware was issued *after* the firmware currently running on the Smart Meter:

$$\begin{aligned}&precedephase(provide(MPO, fw_i(SS_i)),\\ &\qquad flash(SM(fw_j(SS_j)), fw_i(SS_i)),\\ &\qquad V(provide(MPO, fw_j)(SS_j), \emptyset))\end{aligned} \quad \text{(Req3)}$$

Assumptions.

The following set of assumptions will be used for the proof:

1. Message Authentication Code: Whenever a mac of a firmware is verified using a shared secret it must have been generated with the same shared secret by the MPO. (Note that we disregard the investigation of the Shared Secret's confidentiality here and assume it to be confidential.)

   $$\begin{aligned}&precede(sign(MPO, fw_i(SS_i), SS_j, mac_j(fw_i(SS_i)))\\ &\quad verify(SM(fw_j(SS_j)), fw_i(SS_i), mac_j(fw_i(SS_i))))\end{aligned}$$

2. The MPO (i) signs only firmwares provided by itself and (ii) signs only with shared secrets of firmwares that were provided before the firmware to be signed.

   $$\begin{aligned}&precedephase(provide(MPO, fw_i(SS_i)),\\ &\quad sign(MPO, fw_i(SS_i), SS_j, mac_j(fw_i(SS_i))),\\ &\quad V(provide(MPO, fw_j(SS_j)), \emptyset))\end{aligned}$$

3. The Measuring Point Operator will only deploy Smart Meters with firmwares provided by itself.

   $$\begin{aligned}&precede(provide(MPO, fw_i(SS_i)),\\ &\qquad deploy(MPO, SM(fw_i(SS_i)))\end{aligned}$$

4. The Smart Meter can only boot the firmware that it was deployed with or that has been flashed.

   $$\begin{aligned}&precede(\{deploy(MPO, SM(fw_i(SS_i))),\\ &\qquad flash(SM(fw_j(SS_j)), fw_i(SS_i))\},\\ &\quad boot(SM, fw_i(SS_i))\end{aligned}$$

5. A Smart Meter running an approved firmware will verify a newly received firmware's MACs with the current shared secret before flashing it. The set $provfw$ denotes the set of firmwares that were actually provided by the MPO.

   $$\begin{aligned}&\forall fw_j \in provfw :\\ &\quad precede(verify(SM(fw_j(SS_j)), fw_i(SS_i),\\ &\qquad mac_j(fw_i(SS_i))),\\ &\qquad flash(SM(fw_j(SS_j)), fw_i(SS_i)))\end{aligned}$$

6. The execution of an action by the Smart Meter under a given firmware fw_i requires the Smart Meter to have booted this firmware:

   $$precede(boot(SM, fw_i(SS_i)), \Sigma_{/SM(fw_i(SS_i))})$$

Proof.

Assumption 2. can be limited to its first aspect (i) to imply the following property:

$$\begin{aligned}&precede(provide(MPO, fw_i(SS_i)),\\ &\quad sign(MPO, fw_i(SS_i), SS_j, mac_j(fw_i(SS_i)))\end{aligned} \quad (0)$$

With this, the proof can be conducted similar to Section 4.1.1's proof using *provide* instead of *approve* and the MPO instead of the AB up to Proof Statement (13).

$$nothappen(\{boot(SM, fw_i(SS_i)) \mid fw_i \notin provfw\}) \quad (12)$$

Using this knowledge we can conclude through Theorem 4 with Assumption 6. that the Smart Meter only performs actions with provided firmwares:

$$nothappen(\{a \in \Sigma_{/SM(fw_i(SS_i))} \mid fw_i \notin provfw\}) \quad (13)$$

Therefore Assumption 5. extends to the whole SM's life cycle

$$\begin{aligned} precede(&verify(SM(fw_j(SS_j)), fw_i(SS_i),\\ &mac_j(fw_i(SS_i))),\\ &flash(SM(fw_j(SS_j)), fw_i(SS_i))) \end{aligned} \quad (14)$$

Through Theorem 1 with Assumption 1. and further Lemma 2 with Assumption 2. it is possible to conclude

$$\begin{aligned} &precedephase(provide(MPO, fw_i(SS_i)),\\ &flash(SM(fw_j(SS_j)), fw_i(SS_i)),\\ &V(provide(MPO, fw_j(SS_j)), \emptyset)) \end{aligned} \quad (15)$$

which is the requirement. *q.e.d.*

Analysis.

- From the formal proof it can be seen that the Sym^2 specification foresees not only the correctness but also the freshness for non-compulsory relevant firmwares. The explicit mentioning of changing the shared secret with every release suggests that the designers were aware of the problem of downgrading and boot cycling, but decided to only specify counter measures in non-compulsory relevant software download.
- Besides this fact, also this mechanism has all the weaknesses of the improved Software Download for compulsory relevant software.
- Additionally, because of the nature of MAC, the confidentiality of the shared secret is very important as it not only allows for verification but also for generation of Message Authentication Codes. An encryption of the firmware during download should be foreseen in order to protect it. Using the shared secret for authenticating encryption schemes could be the easiest solution with fewest alteration.
  However this aspect was not included in the above investigation.

5. CONCLUSIONS

Within this paper, an interesting formal investigation of the SYM^2 Remote Software Download mechanisms was made. As a basis for the formal methods reasoning served the Fraunhofer Secure Modeling Framework (SeMF).
Applied to the relatively young SYM^2 general specification for Smart Electricity Meter Embedded Systems firmware update mechanisms, valuable results were obtained.Both proposed variants for Remote Software Download, the one for compulsory calibration relevant software as well as the one for non-compulsory relevant software showed to roughly meet the defined requirements of correct, fresh origin of the software image.
The compulsory calibration relevant mechanism showed a vulnerability for downgrade and upgrade cycle attacks. Using the "boot cycling" attack, an attacker is able to undermine the availability of the meter. Furthermore, ensuring the secure boot of the Smart Meter is an issue. This becomes clear with the pre-proof assumption, that a Smart Meter can only boot the firmware that is was deployed with or that has been flashed. Experience, especially in the Embedded Systems domain has shown severe security flaws in the boot sequences of different devices. Remedy can be found by using Trusted Computing concepts for boot.

The compulsory calibration relevant Remote Software Download mechanism of SYM^2 can be improved by adding a version number to the software image and checking it against the current firmwares version before flashing. This effect is shown in section 4.1.1 with an improved and extended SeMF system model. Freshness of the firmware is to be guaranteed in this case. But secure boot related issues remain.

For the non-compulsory calibration relevant Remote Software Download mechanism, freshness and correct origin of the firmware image is shown. Compared to the compulsory calibration relevant variant, downgrade or "boot cycling" attacks are not possible. This is founded by the fact of using changing shared secrets, which is slightly comparable with software version numbers embedded in the software images.
The difficulty in assuring a secure boot remains for this variant, too. It may also be solved by adding Trusted Computing functions to the system, enabling an authenticated boot mechanism.
Another weakness is the confidentiality of the shared secret. When transmitting the software image without encryption, it could be eavesdropped by an attacker and analysed for the position of the shared secret.

Using formal methods reasoning for Software Download concepts of Embedded Systems is an interesting and valuable approach. This is not limited to the Metrology domain and can also be applied to other Embedded Systems domains, such as Automotive or Industry Control.

6. REFERENCES

[1] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8, 1990.

[2] S. Eilenberg. *Automata, Languages and Machines.* Academic Press, New York, 1974.

[3] European Parliament. Directive 2004/22/ec of the european parliament and of the council of 31 march 2004 on measuring instruments. http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?-uri=OJ:L:2004:135:0001:0080:EN:PDF, 3 2004.

[4] A. Fuchs, S. Gürgens, D. Weber, C. Bodenstedt, and C. Ruland. Formalization of smart metering requirements. In *Proceedings of S&D4RCES 2010 Workshop, in the Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems (SAFECOMP 2010).* ACM DL Digital Library, 2010.

[5] R. Grimm and P. Ochsenschläger. Binding Cooperation, A Formal Model for Electronic Commerce. *Computer Networks*, 37:171–193, 2001.

[6] S. Gürgens, P. Ochsenschläger, and C. Rudolph. Authenticity and Provability – a Formal Framework. GMD Report 150, Fraunhofer Institute for Secure Telecooperation SIT, 2002.

[7] S. Gürgens, P. Ochsenschläger, and C. Rudolph. Parameter confidentiality. In *Informatik 2003 – Teiltagung Sicherheit*. Gesellschaft für Informatik, 2003.

[8] S. Gürgens, P. Ochsenschläger, and C. Rudolph. On a formal framework for security properties. *International Computer Standards & Interface Journal (CSI), Special issue on formal methods, techniques and tools for secure and reliable applications*, 27(5):457–466, June 2005.

[9] OIML. Oiml d 31 general requirements for software controlled measuring instruments. http://www.oiml.org/publications/D/D031-e08.pdf, 2008.

[10] Physical-Technical Federal Institute of Germany (PTB). Requirements for software-based metering devices. http://www.ptb.de.

[11] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[12] SYM2-project. Functional specification document for electricity metering devices. http://www.sym2.org.

[13] WELMEC. Welmec 7.2 issue 4 software guide. http://www.welmec.org/ /fileadmin/user_files/publications/7-2_Issue4.pdf, 5 2009.

[14] WELMEC. Welmec geography. http://www.welmec.org/welmec/welmec-tour/welmec-geography.html, 12 2009.