Information Technology

# Software-Sicherheit in virtualisierten Infrastrukturen – Das Beispiel der intelligenten Stromzähler –

# Software Security in Virtualized Infrastructures – The Smart Meter Example –[1]

**Prof. Dr. Bernhard Beckert:** Institute for Theoretical Informatics – KIT, Am Fasanengarten, 5, 76131 Karlsruhe, Germany
Tel: +49 721 608 44025, Fax: +49 721 608 53088, E-Mail: beckert@kit.edu

**Jun.-Prof. Dr. Dennis Hofheinz:** Institute for Cryptography and Security – KIT, Am Fasanengarten, 5, 76131 Karlsruhe, Germany
Tel: +49 721 608 45271, Fax: +49 721 608 55022, E-Mail: dennis.hofheinz@kit.edu

**Prof. Dr. Jörn Müller-Quade:** Institute for Cryptography and Security – KIT, Am Fasanengarten, 5, 76131 Karlsruhe, Germany
Tel: +49 721 608 44205, Fax: +49 721 608 55022, E-Mail: muellerq@ira.uka.de

**Prof. Dr. Alexander Pretschner:** Institute for Programming Structures and Data Organisation – KIT, Am Fasanengarten, 5, 76131 Karlsruhe, Germany
Tel: +49 721 608 45080, Fax: +49 721 608 55079, E-Mail: alexander.pretschner@kit.edu

**Prof. Dr. Gregor Snelting:** Institute for Programming Structures and Data Organisation – KIT, Adenauerring, 20a, 76131 Karlsruhe, Germany
Tel: +49 721 608 44760, Fax: +49 721 608 58457, E-Mail: gregor.snelting@kit.edu

**Keywords:** virtualized infrastructures, E-Energy, smart meters, software security, homomorphic encryption, proof-carrying code, information flow control, deductive verification, runtime verification

**Schlagworte:** virtualisierte Infrastrukturen, E-Energy, intelligente Stromzähler, Software-Sicherheit, homomorphe Verschlüsselung, beweistragender Code, Informationsflusskontrolle, deduktive Verifikation, Laufzeitverifikation

| | | |
|---|---|---|
| **MS-ID:** | gregor.snelting@kit.edu | February 2, 2011 |
| Heft: 53/* (2011) | Received: 1 Accepted: 2 | |

[1] An extended version of this overview article can be found in the technical report 2010-20, Department of Informatics, KIT, 2010.

**Abstract**

Future infrastructures for energy, traffic, and computing will be virtualized: they will consist of decentralized, self-organizing, dynamically adaptive, and open collections of physical resources such as virtual power plants or computing clouds. Challenges to software dependability, in particular software security will be enormous.

We use the example of smart power meters to discuss advanced technologies for the protection of integrity and confidentiality of software and data in virtualized infrastructures. We show that approaches based on homomorphic encryption, proof-carrying code, information flow control, deductive verification, and runtime verification are promising candidates for providing solutions to a plethora of representative challenges in the domain of virtualized infrastructures.

**Zusammenfassung**

Zukünftige Infrastrukturen für Energie, Verkehr und Computing werden virtualisiert sein: sie werden aus dezentralisierten, selbstorganisierenden, dynamisch adaptiven und offenen Verbänden physischer Resourcen wie etwa virtuelle Kraftwerke oder Computing Clouds bestehen. Die Herausforderungen an Verlässlichkeit der Software, insbesondere an Software-Sicherheit werden enorm sein.

Wir diskutieren am Beispiel intelligenter Stromzähler zukünftige Technologie zum Schutz von Integrität und Vertraulichkeit von Software und Daten in virtualisierten Infrastrukturen. Wir zeigen, dass homomorphe Kryptographie, beweistragender Code, Informationsflusskontrolle, deduktive Verifikation und Laufzeitverifikation ein hohes Potential haben, Lösungen für eine Fülle von Herausforderungen im Bereich der virtualisierten Infrastrukturen zu liefern.

## 1 Introduction

Future infrastructures for energy, traffic, and computing will be *virtualized*, and will depend on software to an unprecedented amount. Virtual power plants will consist of dynamically adaptive, heterogeneous collections of physical power sources such as wind power generators or photovoltaic panels. Traffic management will rely on large-scale simulation and multi-modal route planning; future trips will happen in a virtual environment before they take place in the physical world. Cloud computing – the prototype of a virtualized infrastructure – provides computing power through Internet outlets.

Hence, future infrastructures will depend on software to an amount previously unimaginable. And while the state of the art perhaps allows to develop the necessary software functionality, virtualization generates *software dependability* problems, which cannot be handled by today's software technology. Dependable functionality, communication, fault tolerance, adaptivity, safety, security, and privacy will not only require the adaption of known dependability techniques, but also the development of new ones. For example, model checking or verification have never been applied to self-organizing software driving virtual power plants.

Software security will pose a particular challenge in virtualized infrastructures. Recent attacks, e.g., based on the Stuxnet worm, on SCADA (Supervisory Control and Data Acquisition) systems controlling industrial infrastructure demonstrate that even today, security is fragile. This problem will multiply in virtualized infrastructures. Thus, *integrity* will be essential, meaning that input, output, and critical computations cannot be manipulated from outside. For the protection of privacy, *confidentiality* will be essential (meaning that private or secret data cannot flow to unauthorized recipients), as well as appropriate filtering and aggregation of data. Classical IT security techniques such as access control and encryption will need additional breakthroughs, such as homomorphic cryptography, to be useful in energy or traffic infrastructures. New techniques such as semantics-based software security analysis and information flow control will be needed to master integrity and confidentiality challenges.

In this overview article, we investigate software security problems in future virtualized infrastructures; using smart metering as an example. We demonstrate how advanced security technologies will be able to protect integrity and privacy. We concentrate on particularly promising techniques, namely homomorphic cryptography, information flow control, deductive verification, proof-carrying code, and runtime verification. We indicate how these techniques can be used to protect other components in critical infrastructure, such as SCADA systems. Note that we present a design for a future security toolbox, but not (yet) specific results.

## 2 Smart Metering Systems

### 2.1 Background

Smart metering technology makes it possible to continuously measure the consumption of energy, gas, and water. Because the measuring devices are directly connected to a respective IT infrastructure, it is possible to transmit the measurement data in varying intervals to a piece of data administration software ("cockpit") which runs on a PC in the respective household or company, or directly to the energy provider or billing company.

The advantages are considered manifold: there is no need for physical people to read the meters; households can themselves detect a potential waste of energy by continuously monitoring consumption; fine-grained consumption information allows energy providers to tune the load balancing of their networks; since resources cost differently at different times, households can automatically switch on, say, washing machines at the cheapest moment of the night.

Whether or not all these anticipated advantages will become reality is not the subject of this paper: for instance, we do not discuss if the energy used for a continuously running DSL modem does not outweigh the saved energy (which is estimated to not exceed ca. Euro 3,00 per month per household); nor do we discuss if load balancing should continue to be done at the street level; nor do we touch the legal perspective (see, e.g., [15]).

We are convinced, however, that smart metering systems are an excellent example for the convergence of business and embedded IT and therefore are highly representative of tomorrow's virtualized infrastructures. Moreover, it is a fact that there is a politically motivated desire to install these devices on a large scale; that in terms of smart meters for electricity, a regulation (2006/32/EC) requires new houses to be equipped with respective basic technology for energy efficiency reasons as of January 2010, and that consumption data must be transmitted electronically in standardized form since April 2010; that, following European legislation, the German Energy Industry Act (Energiewirtschaftsgesetz, EnWG) requires the unbundling of energy providers, measurement device operators, and device readers; and that major energy providers are running huge sets of test installations today.

On the other hand, the economic benefits of rolling out smart metering technology remains to be proved; information security problems that are concerned with the measuring devices as well as with communication of the measurement data have not fully been solved yet; and it is also true that the population is becoming increasingly aware of the potential privacy issues that emerge from this innovative technology, as highlighted by the example of the 2008 Big Brother award to Yello Strom for their smart metering technology.

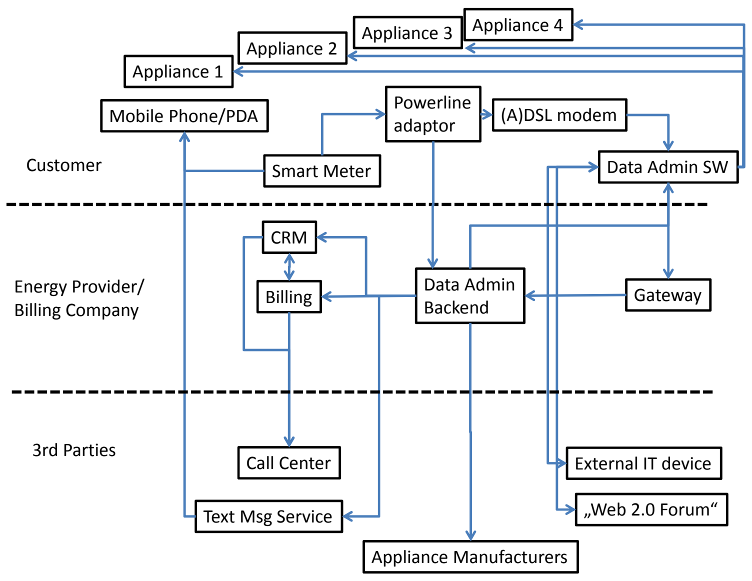


Figure 1: Smart metering systems: Bird's eye view

## 2.2 Architecture of a Typical Smart Metering System

Energy is measured in the measuring device, which sends the data to a data concentrator ("MUC"); both devices together are usually called the smart meter. Data administration software is used to check the current consumption, to build personal profiles, and to contrast these personal profiles to other profiles (see below). In order to reduce the attack surface, it seems of course advisable to physically implement data management and control of appliances in separate devices.

Text messaging and email services are being implemented that warn members of a household if they have likely forgotten to switch off, say, an oven. Metering data can be sent from the data administration software to many other IT systems, including Web 2.0 services such as social networks. Conversely, parts of the data admin software can be implemented in the cloud so that access via external PCs becomes possible.

When data is transmitted from the household to the energy provider or the respective billing company, a plethora of IT systems enters the game. These include gateways for the metering data, a web back-end for the end customer's data administration software that, among other things, can provide profiling data of comparable households, billing services, CRM systems, the implementation of sending the above warning text messages or emails, etc. Finally, it is perfectly conceivable that in case customers agree, their data is sent to third parties, including appliance vendors, call centers, advertisement providers, marketing companies, etc.

A typical architecture of the overall system – of which every energy provider of course offers differing instantiations – very roughly looks as depicted in Figure 1 (boxes are components, arrows represent data flows).

## 2.3 Trusted Device

For security reasons, certain components of the smart metering system must be physically protected from manipulation. In particular, the measuring device itself must be protected from physical manipulation to ensure that the measurement corresponds to the true electricity consumption; there must be a *trusted device* providing certain (software) functionality, including encryption, which is protected from manipulation of its software; finally, the devices that physically switch appliances must be protected from manipulations.

In our architecture, we assume that the smart meter (i.e., the measuring device and the MUC) and the trusted device are the same logical component. Software with different trust levels runs on the trusted device (core, kernel, application). The core cannot be updated remotely. The kernel can be updated but only from trusted sources. The applications can come from the same source as the cockpit software.

## 3 Challenges

In a smart metering environment as described above, a number of challenges arise, both related to the integrity and confidentiality of software and data. Concretely, we can isolate several desirable properties of a smart metering system.

First of all, a customer might be interested in protecting his or her detailed power consumption traces: Individual electrical devices (ovens, hair dryers, TV sets, etc.) have characteristic power consumption patterns which make it possible to even identify single appliances [18]. Hence, detailed power traces are useful for marketing purposes. For instance, heavy users of kitchen devices are more likely to be susceptible to food-related advertisements. Heavy computer users might be more susceptible to advertisements for microelectronics or computer games.

One could also analyse power consumption patterns to identify individuals from certain groups (e.g., students, jobless persons, night-shift workers, etc.). Besides, power traces can be used to determine, e.g., how many people live in the household, when the household members are on vacation, or even when they leave the house. In principle, this data is useful for burglars.

Hence, to protect the customer's privacy, detailed power consumption traces should be protected [18]. Of course, on the other hand, the energy provider has a legitimate interest in using power consumption information for billing and to predict power demands and adjust its infrastructure.

Furthermore, the integrity of the system and, in particular, the trusted device must be protected from attacks. For this, the design and the correct implementation of the software in the trusted device plays a central rôle. As a smart meter will be installed in households for quite some time before they are exchanged, it should be possible to remotely update the software on the trusted device. It is a difficult challenge to nevertheless ensure integrity. The core of the trusted device, which cannot be updated itself, has to provide this assurance.

Measurements exist both in raw and in aggregated form. These aggregations pertain to the dimensions of both time (seconds, hours, days, months) and space (one appliance, a household, a house, a block, a district, a city). Among other things, whenever these aggregations are used for control purposes, e.g., load balancing, their integrity and authenticity become crucial properties. Otherwise, a possible attack consists in tricking an energy provider into thinking that either too much or too little energy will be needed at a specified moment in time, with potentially hazardous consequences for the infrastructure.

Finally, one has to ensure that control signals are not tampered with. Even if they are generated by the cockpit or the user via PDA they cannot be trusted completely. Terrorists could start a distributed denial-of-service attack or worse if they can install malware on the cockpit and thus switch on a large number of appliances at the same time, producing a surge in energy consumption and system breakdown. The only protection is that switching is done by the trusted device (possibly requiring authorisation from the provider for certain changes in consumption).

## 4 Homomorphic Encryption

In this section, we will outline techniques to *securely* and *efficiently* aggregate data using homomorphic encryption. This will in particular be useful to our secure metering use case. However, of course the techniques will be versatile enough for more general applications.

In smart metering, we will only need to operate in a very specific way on encrypted data. More specifically, we will only need an *additively homomorphic* encryption scheme, which allows to compute the encryption of the *sum* of several encrypted plaintexts.

Paillier's encryption scheme [19] is an example of an additively homomorphic encryption scheme. A distinguishing feature of Paillier's encryption scheme is the (additively) homomorphic property: we have

$$\mathsf{Enc}(pk, m_1) \cdot \mathsf{Enc}(pk, m_2) = \mathsf{Enc}(pk, m_1 + m_2)$$

### 4.1 Applications to Smart Metering

In a smart metering system, the aggregation of measurements before transmitting them to the energy provider, will ensure confidentiality of the customer's detailed power traces. Aggregation can happen in two dimensions: over time (i.e., we can aggregate measurements from throughout the week), or space (i.e., we can aggregate from several customers). In both cases, only an additively homomorphic encryption scheme is necessary. Garcia and Jacobs [10] explain how the secure aggregation of measurements across several customers can be performed using an additively homomorphic encryption scheme such as Paillier's scheme (other approaches in the context of billing and load distribution also exist [5, 4, 20]). Concretely, the idea is as follows: Each customer $i$ performs his or her own measurement $m_i$. The goal is to compute the aggregation $\sum_i m_i$ of several measurements from several customers. Each customer possesses a Paillier public/secret keypair, as does the energy provider. All customers engage in an efficient multi-party protocol to compute an encryption of the aggregation $\sum_i m_i$ of measurements under the energy provider's public key. This requires only a link from each customer to the energy provider. In the end, the energy provider decrypts the aggregated data and only learns the aggregation of measurements, while no customer learns anything about other customers.

This approach of secure aggregation demonstrates the applicability of (limited) homomorphic encryption to the smart metering setting. In particular, cryptography can be used to simultaneously achieve seemingly contradictory requirements (the energy provider's desire to gather information vs. the customer's privacy) [10]. Our goal is to extend these ideas for the use in a practical smart metering system. In particular, it is a unique challenge to combine aggregation and homomorphic cryptography with digital signatures (which are an important instrument for integrity of smart meters).

### 4.2 Fully Homomorphic Encryption

Essentially, additively homomorphic encryption schemes only allow the summation (and hence averaging) of encrypted data. This property is useful in the smart metering example, and can be efficiently achieved, e.g., using Paillier's encryption scheme. However, certain scenarios call for more general homomorphic properties of encryption schemes.

For instance, a *fully* homomorphic encryption (FHE) scheme allows arbitrary computations on encrypted data. Until very recently, *fully homomorphic* encryption schemes were actually deemed impossible. However, in a breakthrough work, in 2008 Craig Gentry from the IBM T.J. Watson research center finally succeeded in constructing the first FHE scheme [11].

Fully homomorphic encryption might seem like the obvious way to achieve secure cloud computing: instead of sending all data in plain into the cloud to outsource computations on that data, *encrypt* all data, and let the cloud compute on this encrypted data. The encrypted result can then be sent back to the customer, who possesses the secret key to decrypt the result. But as of today, FHE schemes are far too inefficient to be directly useful in the cloud computing setting. That is, computing on encrypted data is far more expensive than computing on unencrypted data. Current implementations of FHE schemes may require several seconds to perform a single logical operation on encrypted data.

Hence, while FHE schemes offer vast possibilities, additional research is required to fit FHE schemes to applications. One current effort in the cryptographic community is thus speeding up current FHE schemes through algorithmic improvements. Furthermore, in several applications (e.g., secure multiparty computations), FHE schemes can be supported by existing solutions, e.g., secure hardware. We can hope to get "the best of both worlds," i.e., both the functionality of FHE schemes and the efficiency of existing solutions.

## 5 Language-Based Security

Traditional software security mechanisms, such as access control, certifications of origin, protocol verification, intrusion detection, will of course be necessary in virtualized infrastructures, but will not be sufficient. For smart metering, *integrity* will be essential, meaning that critical computations cannot be manipulated from outside. For privacy, *confidentiality* will be essential, meaning that private data cannot flow to public ports.

Research in software security has developed techniques such as proof-carrying code and information flow control (IFC), which analyze the true semantics of software, and provide guarantees about software behavior and not just its "packaging". As such analyses examine the program source code, they are called "language-based". Experimental security infrastructures based on these techniques have been developed in large European projects [7]. Modern program analysis based on interprocedural dataflow analysis, abstract interpretation, or model checking has developed powerful tools for discovering anomalies in software. IBM developed an IFC tool which can analyse large programs written in full Java [8]. New results concerning central notions such as noninterference and declassification are pursued in the new DFG priority program "Reliably Secure Software Systems" (RS3). RS3 integrates software security with advanced verification and program analysis.

### 5.1 Proof-carrying Code

Proof-carrying code comes with an (encoded) formal proof of some desireable property. Properties might be functional, safety, or security related. Proofs are written in some formal logic, and refer to the program text of the software (e.g., loop invariants in Hoare logic). Upon installation, the proof must automatically be checked for correctness, and it must be checked that the proof does indeed correspond to the software component. Proof-carrying code is based on the fact that checking a proof can be done efficiently, in contrast to the expensive (manual) construction of the proof. In the literature, proof checkers have been described in detail. The European project Mobius has developed a security infrastructure based on proof-carrying code, which is used for Java code in mobile devices.

In the smart meter, proof-carrying code could be very helpful once new software versions are downloaded to the smart meter. Integrity and privacy properties must be formalized when developing the software to be downloaded, and corresponding formal proofs be constructed (this will be a nontrivial task). The checker is based on theorem prover technology, and must be part of the trusted device (see Section 2.3). Upon download, the checker will guarantee functionality and security, or – if proof checking fails – it will disallow installation.

Note that this approach assumes a fixed (formal) specification of the functionality to be downloaded. If the specification changes, the checker can still check whether the code satisfies it, but cannot check the validity of the new specification. In such cases, the formal specification must be supplied or downloaded independently and rely on certification keys.

### 5.2 Information Flow Control

As an alternative to proof-carrying code, new IFC techniques can be applied to guarantee integrity and privacy. Data which is marked confidential (e.g., power consumption traces) must not flow to public ports (e.g., the gateway of the energy provider), or perhaps only in aggregated form as discussed in Section 4. Similarly, critical computations (e.g., appliance switching commands) must not be manipulated from outside (e.g., by

```
class PasswordFile {
 private String[] names;
                /* P: confidential*/
 private String[] passwords;
                /* P: secret*/
 // Pre:all strings are interned
 public boolean check(String user,
   String password /*P: confidential*/) {
   boolean match = false;
   try {
   for (int i=0; i<names.length; i++) {
    if (names[i]==user
      && passwords[i]==password) {
      match = true;
      break;
    }
   }
  }
  catch (NullPointerException e) {}
  catch (IndexOutOfBoundsException e) {};
  return match;  /* R: public*/
  }
}
```

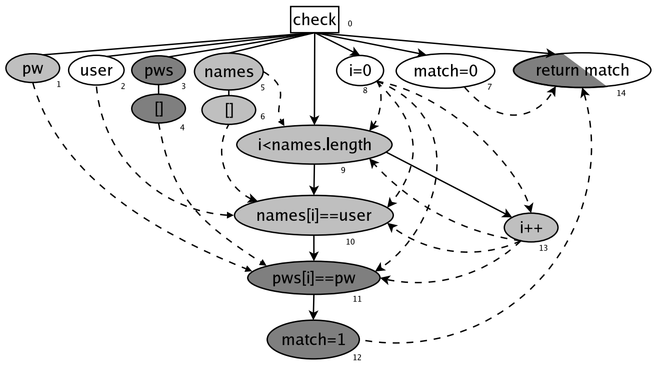


Figure 2: A simple Java password checker and its program dependency graph (without exceptions) with computed security levels (white = public, grey = confidential, dark = secret). The program contains a security leak showing up as a level conflict in the return node (upper right).

the billing company – but perhaps manipulation from the "cockpit" is allowed).

Technically, information flow control is difficult, in particular for realistic programs (e.g., 100 kLOC) written in realistic languages (e.g., full Java byte code). Concurrency and multi-threading make information flow particular demanding. The theoretical foundations, such as noninterference and declassification, are still subject to ongoing research. The Mobius project delivered the first information flow infrastructure for Java Card applications on mobile devices; it is based on security type systems. In Germany, RS3 integrates information flow control with modern program analysis and verification technology.

A precise IFC analysis must exploit flow-sensitive, object-sensitive, and context-sensitive information as computed by interprocedural dataflow analysis. The results of such an analysis can be encoded in form of a program dependency graph, as indicated in Figure 2. Without going into details, note that information can flow in the program only along paths in the dependency graph. If there is no path, it is guaranteed that there is no (illegal) flow of information. This fundamental property (for which a machine-checked formal proof exists [22]) makes dependency graphs so suitable for information flow control. Note that in the presence of procedures, arrays, objects, exceptions, etc. the construction of the graph becomes very complex. Today, two dependency graph implementations for full Java exist (one, the JOANA tool, developed at KIT), as well as a commercial implementation for C/C++, called CodeSurfer. JOANA can handle full Java bytecode and scales up to 50 kLOC. Full details can be found in [13, 12].

For smart meters, IFC, together with other program analysis methods (see e.g. [21]), can guarantee that integrity of the trusted device cannot be broken by software attacks; which is essential for dependable cryptography and proof checking. Information flow control can also guarantee that household appliances cannot be controlled directly by external software, thus protecting safety and integrity of the appliances. Information flow control will guarantee privacy protection by introducing appropriate security levels for secret, encrypted, aggregated, and public data; and analysing the information flow for all such data in the smart meter and the cockpit. Analysis must carefully introduce declassification [17] e.g., at aggregation points.

## 6 Deductive Program Verification

Functional correctness of the trusted device's kernel is so central for integrity of the smart metering system that a formal verification is certainly justified. Formal verification is needed in particular for the hardware interface (measuring device and appliance switching), cryptographic services, the proof checker built into the trusted device, information flow control implementations, and communication/authentification services.

Fortunately, program verification today can be applied to real-world software. The Verisoft XT project showed that verifying an operating system micro kernel is feasible [3]. It verified properties of SYSGO's PikeOS, which may very well serve as the basis for implementing a trusted device kernel for an advanced smart meter implementation. Another spectacular verification is the L4.verified project [6]. Such achievements clearly demonstrate that verification of the trusted device is indeed possible.

Besides guaranteeing functional correctness, verification can be used to formulate information-flow problems as proof obligations in program logics. We can leverage these advances together with our own experience in formal methods for functional properties in order to specify and verify information flow properties.

In the simplest case, a confidentiality policy can be formalized as non-interference and described in terms of an indistinguishability relation on states. That is, two program states are indistinguishable for $L$ if they agree on values of $L$ variables. The non-interference property says that any two runs of a program starting from two initial states indistinguishable for $L$, yield two final states that are also indistinguishable for $L$ variables.

In a smart-metering system, more complex properties such as controlled information release need to be assured. We plan to define syntax and semantics of a specification language for information-flow properties at the level of (Java) programs. The goal is a language that is expressive enough to allow security requirements at the system level to be easily and flexibly broken down into program level requirements. Further, we will design and implement a system for verifying programs annotated with security properties and specifications. More specifically, we will be concerned with the rule-based generation of first-order verification conditions from annotated Java programs. The technological basis will be the KeY system (co-developed at KIT) [2].

Our project is based on recent advances in using program logics (such as Hoare Logic or Dynamic Logic) for the specification and verification of information-flow properties at code level. Using program logics, non-interference can be directly formalized; or it can be translated into dependence properties, which in turn can be formalized in program logics. Non-interference can also be translated into proof obligations that can – in principle – be handled by unmodified existing program verification tools using a technique called *self-composition* [1].

We also plan to adapt the concept of *ownership* to the verification of information-flow properties. This concept has been developed in the context of deductive verification of functional properties to specify that complex data structures are not *changed* in unexpected ways (e.g. [16]). For information-flow properties, ownership has to be adapted so that one can specify that data structures are not *read* in an unintended way.

## 7 Runtime Verification

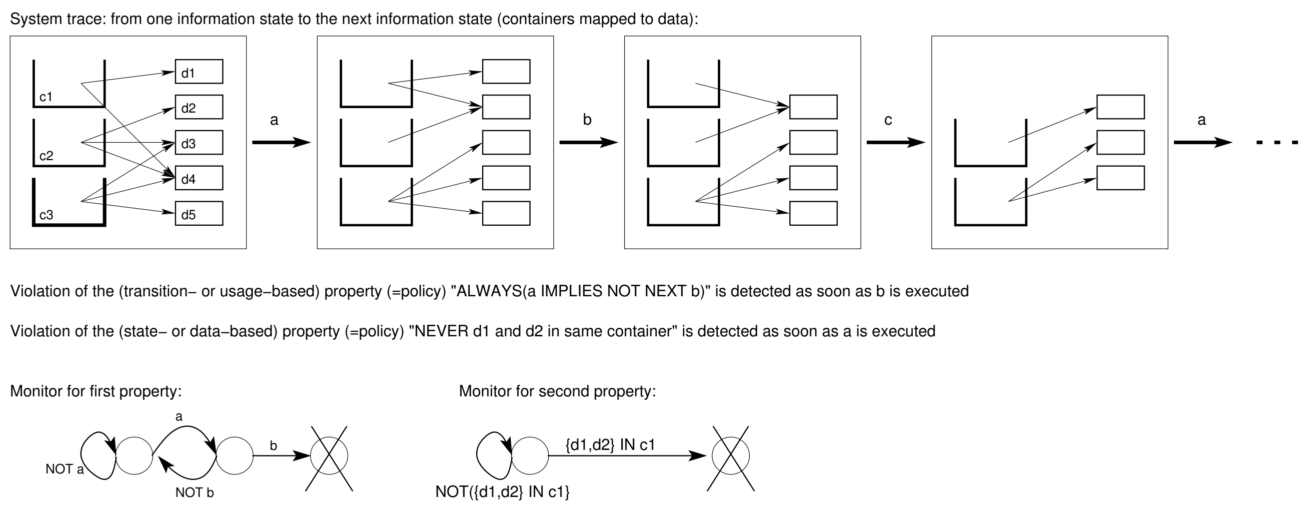
The system architecture in Figure 1 depicts several data flows some of which are potentially privacy-sensitive and deserve protection. The data types in question include raw sensor data, profiles, and customer master data, but also traffic data that is created whenever the customer interacts with any other of the various stakeholders. The problem, then, is to make sure that these different kinds of data are used w.r.t. laws and regulations, but also w.r.t. customer-defined requirements.

As an example, data is collected by the smart metering device and sent to the customer's data management system on a per-second basis, and to the frontend of the energy provider on a 15-minutes basis. The data management software computes profiles, deltas with other people's profiles and historical data, and displays the result of these computations in graphical form. Because the customer has provided his consent, this fine-grained measurement data is sent to a vendor of appliances who can recommend some energy class A fridge. At the same time, the customer may not fully understand his monthly bill and contact a call center which, in turn, has access to a plethora of different kinds of data. In this setting, there are different kinds of data in different representations on different machines in different governance (and liability domains). The problem then is, how can this data be controlled. This is a real problem: Among other things, only recently, a variety of Android mobile phone applications – that could be part of the smart metering system – have been shown to disclose location information to advertisement servers or SIM and phone numbers to other stakeholders without explicitly asking for the user's consent [9].

Roughly speaking, runtime verification denotes decision procedures for whether a future or past temporal logic formula is satisfied, open, or violated for a finite prefix of a possibly infinite trace of (sets of) events. As such, runtime verification is, in contrast to model checking or deductive theorem proving, a technique that is solely used dynamically. Statements on the truth value of a formula are hence made for one given trace and one moment in time rather than for all traces of the system under consideration.

Runtime verification is relevant in the context of smart metering contexts when it comes to monitoring the usage of data. Roughly, monitors are implemented that listen to the events that happen in the system. These events include the access to possibly sensitive data items, copying these items, but also deletion requirements. These events happen at different levels of abstraction, including the level of machine language, data bases, runtime systems such as .NET or Java virtual machines, infrastructure applications such as X11, within applications such as those in Microsoft Office, etc. For each of these layers, events that relate to sensitive data items must be observed. This is done by (automatically) transforming the temporal logic formulas that specify adequate data usage into respective monitors at the respective layers of abstraction.

For controlling data usage, a simple temporal logic with abstractions for limited cardinality constraints is the Obligation Specification Language, or OSL [14]. Traces are sequences of sets of events. Then, given an OSL formula $\varphi$ and a trace (prefix) $t$, runtime verification decides at runtime, for each moment in time $n$, whether or not $\varphi$ is true at $n$ (can never be violated in the future), violated (can never become true in the future), or whether this decision cannot be taken yet. It is possible to automatically synthesize monitors from policies written in OSL. These generated monitors allow us to detect runtime violations of properties like those described in Section 3. With minor extensions, it is in many cases also possible to prevent a policy violation.



System trace: from one information state to the next information state (containers mapped to data):
c1
c2
c3
d1
d2
d3
d4
d5
a
b
c
a
Violation of the (transition− or usage−based) property (=policy) "ALWAYS(a IMPLIES NOT NEXT b)" is detected as soon as b is executed
Violation of the (state− or data−based) property (=policy) "NEVER d1 and d2 in same container" is detected as soon as a is executed
Monitor for first property:
NOT a
a
NOT b
b
Monitor for second property:
{d1,d2} IN c1
NOT({d1,d2} IN c1}

Figure 3: Runtime Verification meets Data Flow Detection

OSL, augmented by constructs to speak of a system's data state, enables to specify policies that allow or disallow the flow of information (1) within one layer of the system, (2) across layers of abstraction, and (3) in-between different systems. This can formally be captured by containers that may or may not contain specific data items. Because OSL can be expressed in LTL, it is almost trivial to automatically derive generic monitors from usage control policies. In order to be applied to the smart metering system, we need to connect these generic monitors to the concrete different subsystems, thus yielding a controlled system where it is possible to detect or prevent the flow of data from, say, the data management software, to, say, a call center. Figure 3 shows an example: the system evolves over time. In each step, the mapping from containers (files or emails or data-base records stored in one of the subsystems) to data (e.g., usage profiles) may change. Policies that are transformed into monitors (state machines) make sure that, at runtime, specific usage patterns or specific mappings from containers to data are disallowed.

For all the different systems that taken together make the overall smart metering system, we need to either write or generate OSL policies to configure the generic runtime monitors that implement usage control and data flow detection. Some of the monitors (or submonitors at one layer of abstraction) are likely to leverage static results from the work on language-based security and static verification. Once such a system is in place, we can provide guarantees in terms of system-wide data flows in the overall distributed smart metering system, thus addressing the important privacy challenges described in Section 3.

## 8 Conclusions

The recent Stuxnet attacks on SCADA systems controlling industrial plants demonstrate that the software security risk is high for today's critical infrastructures. It will be even higher for tomorrow's virtualized infrastructures such as E-Energy, E-Traffic, and Cloud Computing. In this overview, we have described a mix of techniques which will reduce security and privacy risks in such infrastructures. Concentrating on smart metering, we have shown:

- Homomorphic encryption schemes, as well as their combination with authentification methods, allow E-Energy providers to collect usage profiles in aggregated form, while customer privacy is still protected.
- Language-based security methods analyse the true semantics of smart metering software, instead of just providing guarantees about its origin.
- Proof-carrying code allows to securely download software into the smart meter while checking its functionality. The necessary proof checker (as well as the encryption software) resides in a trusted device inside the smart meter.
- Information flow control protects critical computations, such as control of household appliances, and discovers privacy leaks. IFC is also used to protect integrity of the trusted device.
- Deductive verification can guarantee functional correctness for, e.g., the proof checker and the encryption software, as well as for the smart meter kernel. Verification can as well support IFC.
- Runtime verification can dynamically detect illegal information flow in case static IFC is not pos-

sible or too imprecise, or system boundaries need to be crossed.

Attacker models, social engineering, and similar aspects are important as well, but were not discussed here for reasons of space. While we have concentrated on the smart metering example, let us conclude with an outlook to how our technology will help to prevent attacks on SCADA systems such as the recent Stuxnet attacks:

- Stuxnet used stolen certification keys. This highlights our approach, namely that we need to analyse the true semantics of a program and not just certify its origin. Program analysis and IFC are becoming more powerful every year, and will eventually kill Stuxnet.
- Current SCADA systems lack a trusted device, which would greatly reduce the risk of infiltration.
- Stuxnet relies on a whole set of zero-day exploits. The latter are often based on software bugs or attacks such as buffer overflow attacks. Modern program analysis has developed powerful tools for finding such anomalies.
- Verification, while expensive, can today formally verify realistic systems such as SCADA security cores.
- Proof-carrying code techniques prevent downloading malware, and runtime verification can dynamically discover illegal information flow.

We do not claim that we can prevent Stuxnet with our current box of security approaches. But our techniques will certainly make attacks much more difficult. We thus plan to implement our approach in a large-scale project, which aims at critical infrastructures as a whole, and will support not just smart metering and E-Energy, but also E-Traffic and Cloud Computing.

## References

[1] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, CSFW-17, Pacific Grove, CA, USA*, pages 100–114. IEEE Computer Society, 2004.

[2] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[3] Bernhard Beckert and Michał Moskal. Deductive verification of system software in the Verisoft XT project. *KI*, 24(1), 2010. Online first version available at SpringerLink.

[4] J.-M. Bohli, C. Sorge, and O. Ugus. A privacy model for smart metering. In *Communications Workshops (ICC), 2010 IEEE International Conference on*, pages 1 –5, May 2010.

[5] C. Efthymiou and G. Kalogridis. Smart grid privacy via anonymization of smart metering data. In *Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on*, pages 238 –243, 2010.

[6] Gerwin Klein et al. Sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[7] Gilles Barthe et al. Mobius: Mobility, ubiquity, security. objectives and progress report. In *TGC 2006: Proceedings of the second symposium on Trustworthy Global Computing*, LNCS. Springer-Verlag, 2006.

[8] O. Tripp et al. TAJ: effective taint analysis of web applications. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 87–97. ACM, 2009.

[9] William Enck et al. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010. To appear.

[10] Flavio Garcia and Bart Jacobs. Privacy-friendly Energy-metering via Homomorphic Encryption. In *6th Workshop on Security and Trust Management (STM 2010)*, 2010.

[11] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC 2009)*, pages 169–178, 2009.

[12] Christian Hammer. Experiences with pdg-based ifc. In *Proc. International Symposium on Engineering Secure Software and Systems (ESSoS'10)*, February 2010.

[13] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.*, 8(6):399–422, 2009.

[14] Manuel Hilty, Alexander Pretschner, David Basin, Christian Schaefer, and Thomas Walter. A policy language for usage control. In *12th European Symposium on Research in Computer Security*, pages 531–546, 2007.

[15] M. Karg. Datenschutzrechtliche Rahmenbedingungen beim Einsatz intelligenter Zähler. *Datenschutz und Datensicherheit*, 34(6):365–372, 2010.

[16] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *Proc. ECOOP 2008*, LNCS 3086. Springer, 2004.

[17] Alexander Lux and Heiko Mantel. Declassification with explicit reference points. In *ESORICS*, pages 69–85, 2009.

[18] K. Müller. Gewinnung von Verhaltensprofilen am intelligenten Stromzähler. *Datenschutz und Datensicherheit*, 34(6):359–364, 2010.

[19] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology (EUROCRYPT 1999)*, pages 223–238, 1999.

[20] A. Rial and G. Danezis. Privacy-Preserving Smart Metering. Technical Report MSR-TR-2010-150, Microsoft Research, 2010.

[21] Alexander Simon. *Value-Range Analysis of C Programs.* Springer Verlag, 2008.

[22] Daniel Wasserrab. Backing up slicing: Verifying the interprocedural two-phase Horwitz-Reps-Binkley slicer. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs.* `http://afp.sf.net/entries/HRB-Slicing.shtml`, November 2009.