Karlsruhe Reports in Informatics 2010,20
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Software Security in Virtualized Infrastructures

The Smart Meter Example

B. Beckert, D. Hofheinz, J. Müller-Quade, A. Pretschner, G. Snelting

beckert@kit.edu, dennis.hofheinz@kit.edu, joern.mueller-quade@kit.edu,
alexander.pretschner@kit.edu, gregor.snelting@kit.edu

2010

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Fakultät für **Informatik**

Software Security in Virtualized Infrastructures – The Smart Meter Example –

B. Beckert, D. Hofheinz, J. Müller-Quade, A. Pretschner, G. Snelting

Karlsruher Institut für Technologie (KIT)

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 enourmous.

While the problems in this domain transcend any specific instantiation, 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, deductive verification, information flow control, and runtime verification are promising candidates for providing solutions to a plethora of representative challenges in the domain of virtualized infrastructures.

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, in form of Software-as-a-Service, Platform-as-a-Service, or Infrastructure-as-a-Service.

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 systems controlling electrical grids demonstrate that even today, security is fragile. It is beyond any doubt that this problem will multiply in virtualized infrastructures. In future infrastructures, *integrity* will be essential, meaning that input, output, and the process of 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 public ports), as well as appropriate filtering and aggregation of data such that, e.g., information about energy demand and supply can no longer be linked directly to specific individuals. Classical IT security techniques such as access control and encryption will need additional breakthroughs, such as homomorphic cryptography, to be useful in cloud computing 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.

The cluster initiative "Dependable Software for Critical Infrastructures" (DSCI) will develop new foundations and methods for software dependability in virtualized infrastructures. DSCI focuses on E-Energy, E-Traffic, and Cloud Computing. DSCI will provide guarantees for dependable functionality, communication, fault tolerance, adaptivity, safety, security, and privacy in future infrastructures. A general overview of DSCI can be found in [7].

In particular, DSCI investigates new approaches to software security in virtualized infrastructures, which exploit recent achievements in algorithmics, language-based security, cryptography, and verification technology. DSCI will also build on fundamental results to be developed by the new DFG Priority Programme "Reliably Secure Software Systems" (RS3). Several DSCI researchers are also leading RS3 projects. But note that RS3 is not concerned with critical infrastructures. In general, we are not aware of any report that pinpoints the difficult security problems in such infrastructures. Hence the DSCI contribution, as outlined in the current overview article, can be summarized as follows:

Contribution We investigate software security problems in future virtualized infrastructures; using smart metering as an example. We demonstrate how a toolbox of advanced security technologies, such as homomorphic cryptography, information flow control, deductive verification, proof-carrying code, and runtime verification, will be able to protect integrity and privacy in smart metering systems. We indicate how our toolbox can be used to protect other components in critical infrastructure, such as SCADA systems.

Organization We start by introducing our exemplary problem domain, that of smart energy meters, in Section 2. As a basis for discussion, we present an exemplary architecture of such a system, perfectly aware that any concrete system is likely to differ in specific details. On these grounds, we derive a set of challenges and describe them in Section 3. In the remaining sections, we show how to use different technologies to tackle a selection of relevant problems: Section 4 shows how to use homomorphic encryption for privacy-preserving aggregation of data. Section 6 shows how to use deductive verification to the end of ensuring correctness and absence of undesired information flows. Section 5 tackles the problem of undesired information flows on the basis of language-based approaches. Section 7 builds on these approaches and adds to the static approaches of Sections 6 and 5 a dynamic approach that is based on runtime verification and that explicitly targets information flow across system boundaries. The conclusion discusses more general applications in critical infrastructures, e.g. for SCADA systems. Related work is discussed throughout the text.

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, or at least are planned to be, 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, depending on the perspective of the various stakeholders, 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 and comparing it with other households; fine-grained consumption information allows energy providers to tune the load balancing of their networks; since ressources 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 in turn is estimated to not exceed roughly Euro 3,00 per month per household, using today's technology –; nor do we discuss if load balancing will not continue to be done at the level of entire street blocks; nor do we discuss if local operating networks (LONs) – possibly do not necessitate smart metering technology at all to implement intelligent switching of electrical devices when it comes to the anticipated next generation of smart meters that bidirectionally "communicate" with the devices; nor do we touch the legal perspective [26].

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/EG) 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 the EnWG requires the unbundling of energy providers, measurement device operators, and device readers; and that major energy providers are running huge (e.g., 5000 households in Cologne) 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.

2.2 Architecture of a Typical Smart Metering System

In the following, we sketch the architecture of an abstract, yet typical smart metering system for electricity.

Energy is measured in the measuring device itself. The measuring device sends the data to a data concentrator (also called MUC, multi utility communicator; the name is motivated by the connection of a multitude of measuring devices for gas, water, etc.). Taken together, these two devices are usually called the smart meter. Depending on the frequency of transmittal (and, consequently, the degree of aggregation of the measurements), the meter sends measurement data either directly to a mobile phone or PDA, or via a power line or classical DSL modem (1) to a local PC that runs data administration software and (2) to the gateway of the billing company, that can but need not necessarily be the same as the energy provider (unbundling; some solutions also include the energy providers as intermediaries). The 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 for the back end). We will assume that this software is also used to control appliances [1].

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 (whether or not the smart metering software and hardware alone can detect if specific appliances are switched on is the subject of an ongoing debate [31] – in any case, in conjunction with appliances connected to LONs, this is clearly possible, even if the metering device alone does not provide sufficient information for this task). In any case, remote handling of appliances or radiators in intelligent buildings appears feasible.

Metering data can be sent from the data administration software to many other IT systems, including Web 2.0 services such as social networks where, among other things, people can show off how "green" their household is, or where avatars shrink and grow depending on the energy that has been consumed. 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 that, for instance, may offer class A fridges that are guaranteed to be amortized within a specific period of time, call centers, advertisement providers, marketing companies, etc.

Accordingly, 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).

The entire smart metering system is characterized by two main features. Firstly, it com-

[1]Data management and control of appliances can and should of course physically be implemented in separate devices.

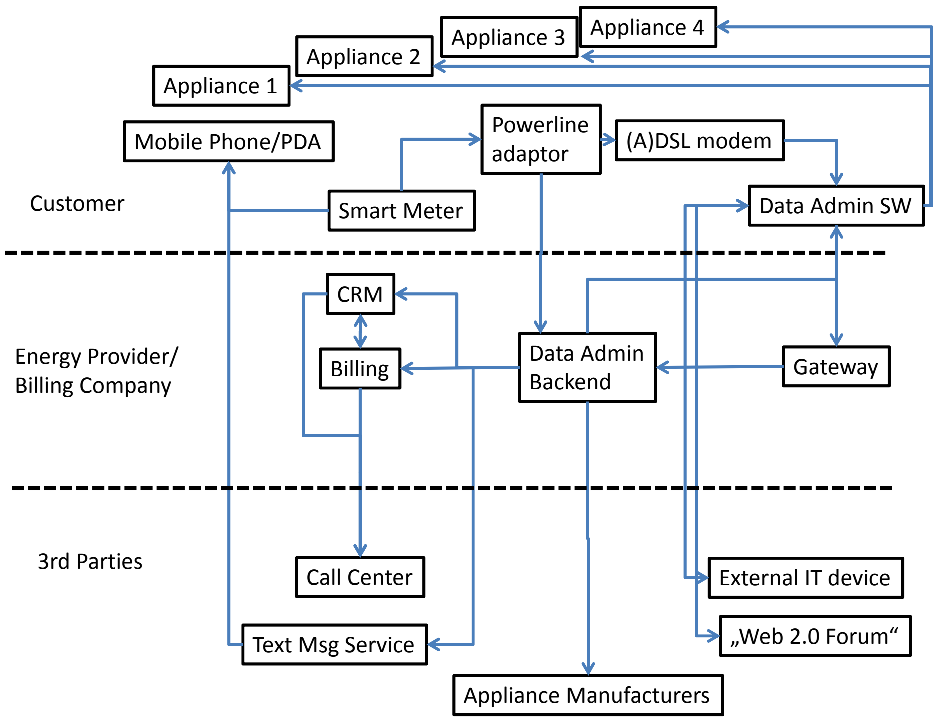


Figure 1: Smart metering systems: Bird's eye view

bines an embedded system (the metering device itself) with several IT systems and, as such, is an excellent example for tomorrow's integrated cyber-physical systems. Secondly, it is a highly distributed system with many different areas of governance, responsibility, and liability: the metering provider and operator, the home cockpit software and its connection to Web 2.0 media, the billing company, the energy provider, and external third parties including call centers and vendors. To date, it is unclear who will be the responsible for the entire system, at least as far as privacy is concerned.

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 eletricity consumption;
- there must be a *trusted device* providing certain (software) functionality, including encryption, which is protected from manipulation of its software;
- the devices that physically switch appliances must be protected from manipulations that may make it accept false switching commands. (These devices may be integral components of the appliances or, for "dumb" appliances they may be part of the socket or plug.)

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.

Architecture of the trusted device. 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.

The kernel part is basically a micro kernel providing functionality with integrity guarantees.

Typical examples for critical functions that may be provided by the trusted device kernel are:

- (secure) communication (in particular with the provider and third parties), information flow limits may be guaranteed for this communication;
- cryptographic services (signing, encryption etc.);
- access to the hardware (measuring device);
- switching appliances, i.e., secure communication with the devices that switch appliances (by signing switching commands);
- getting authorisation from the provider for changes in electricity consumption;
- enforcing upper limits for energy consumption (set by the user or by the provider);
- software updates (this includes checking authenticity of updates, checking proofs in proof-carrying code);
- logging all relevant events.

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.

Confidentiality of customer data. A customer (i.e., the owner of a household) might be interested in protecting his or her detailed power consumption traces. Namely, individual electrical devices (ovens, hair dryers, TV sets, etc.) have characteristic power consumption patterns which make it possible to even identify single appliances [31]. Hence, detailed power traces reveal a level of information about the customer that makes it 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.

Detailed information about power consumption patterns can also be used on a larger scale. For instance, one could match individual power consumption patterns to isolate individuals from certain groups (e.g., jobless persons, night-shift workers, people who arrived home at certain points in time, etc.). Specifically, large-scale data mining could be used for dragnets.

Besides, detailed 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, in particular when such data can be collected and filtered on a large scale. Even more fine-grained data about the household owners can be extracted by matching with typical consumption patterns of, e.g., students, or persons with full-time/part-time job, or without job.

Hence, to protect the customer's privacy, detailed power consumption traces should be protected [31]. 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.

System Software Integrity. The integrity of the system and, in particular, the trusted device must be protected from attacks from the user, the provider, or third parties. For this, the design and the correct implementation of the software in the trusted device plays a central rôle.

As the cockpit software runs on the user's PC on a standard operating system, the integrity of the cockpit software ist hard to protect from attacks by the user (except by obscurity) or by third parties using malware.

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 (otherwise updates are too costly). 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.

Authenticity and integrity of measurements. 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 of tricking an energy provider into thinking that either too much or too little energy will be needed at a specified moment of time, with potentially hazardous consequences for the infrastructure.

Authenticity of control signals. One has to ensure that control signals are not falsified. 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 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).

Certification, trust, and adequacy of requirements. It is not sufficient to build a secure system. Security must be checkable and certifiable. This is particularly important as many stake holders are involved.

4 Fully Homomorphic Encryption: Operating on Encrypted Data

In this section, we will outline techniques to *securely* and *efficiently* aggregate data. 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. Hence, we first introduce the technical tools, and then comment on their use in our smart metering example.

4.1 Fully homomorphic encryption

Motivation: Cloud computing. Virtualized infrastructures such as cloud computing allow to outsource computation tasks through Internet outlets. These Internet outlets are not necessarily trustworthy. In fact, in services such as Amazon's Elastic Cloud, the customer does not even know where the computing outlets are located. In particular when working with sensitive data, it is of course highly undesirable to send all data in plain to an unknown server.

An obvious solution is to encrypt the data before transmitting it to servers in the cloud. However, conventional encryption schemes do not allow to compute on encrypted data: once encrypted, the data can only be decrypted, but not operated on. (In fact, in certain scenarios such as Internet auctions, being able to manipulate encrypted data can become a security weakness: encrypted bids can be modified and then used to overbid a competitor.)

Fully homomorphic encryption (FHE). Until very recently, *fully homomorphic* encryption schemes (i.e., encryption schemes that allow arbitrary computations on encrypted data) 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 [18]. His scheme allows to perform arbitrary computations on encrypted data. The result of such a computation is again encrypted, so that the entity who performs the computation neither learns anything about the data nor about the result.

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.

The (in)efficiency of general FHE. However, there is a catch with this idea. Namely, as of today, FHE schemes are far too inefficient to be useful in the cloud computing setting. That is, computing on encrypted data is computationally far more expensive than computing on plain, unencryted data. Depending on the desired level of security, current (June 2010) implementations of FHE schemes require several seconds to perform a single gate operation (i.e., a bitwise `and`, `or`, or `not` operation) on encrypted data. Besides, due to a highly redundant encoding (in current schemes), encrypting data results in a dramatic blowup in storage requirements. Finally, all operations to be performed on the encrypted data have to be expressed as circuits. In particular, this requires to unroll loops, and follow all branches of `if...then...else...` constructs, which makes the computation in itself much more inefficient.

4.2 Additively homomorphic encryption: an example

Outline. Hence, current homomorphic encryption techniques do not seem ready yet for a direct application to the cloud computing setting. Still, there is hope for practical solutions that only partially rest on the properties of homomorphic encryption. (We will later on comment on such solutions.) Besides, we can still hope for practical solutions that are optimized for specific settings.

Additively homomorphic encryption. In our examples, we will only need to operate in a very specific way on encrypted data. Put differently, we will only need to perform a specific class of homomorphic operations on ciphertexts. More specifically, we will only need an *additively homomorphic* encryption scheme. (That is, an encryption scheme which allows to compute the encryption of the *sum* of several encrypted plaintexts.)

Paillier's scheme. Such encryption schemes are well-known to exist, and in fact are quite efficient. As an example of an additively homomorphic encryption scheme, we recapitulate Paillier's encryption scheme [32]. Paillier's scheme works in the ring $\mathbb{Z}_{N^2}$ for a product $N = PQ$ of two large primes P and Q. Its algorithms are defined as follows:

Key generation. Choose $N = PQ$ and $g \in \mathbb{Z}_{N^2}$ with $\mathrm{ord}(g) = \varphi(N) = (P-1)(Q-1)$. Publish the public key $pk = (N, g)$ and keep the secret key $sk = (P, Q)$.

Encryption. To encrypt $m \in \{0, \dots, N-1\}$, uniformly choose $r \in \{0, \dots, N-1\}$ and compute the ciphertext

$$\mathsf{Enc}(pk, m) = r^N(1+N)^m \in \mathbb{Z}_{N^2}.$$

Decryption. To decrypt $C \in \mathbb{Z}_{N^2}$, compute

$$\begin{aligned} & C^{(P-1)(Q-1)} \\ &= r^{N(P-1)(Q-1)}(1+N)^{m(P-1)(Q-1)} \\ &= (1+N)^{m(P-1)(Q-1)} \\ &= 1 + m(P-1)(Q-1)N, \end{aligned}$$

from which $m(P-1)(Q-1) \bmod N$ and thus m can be computed. (Note here that $1+N$ has order N, since $(1+N)^N = N^2 = 0 \bmod N^2$.)

A distinguishing feature of Paillier's encryption scheme is the (additively) homomorphic property: we have

$$\begin{aligned} & \mathsf{Enc}(pk, m_1) \cdot \mathsf{Enc}(pk, m_2) \\ &= r_1^N(1+N)^{m_1} \cdot r_2^N(1+N)^{m_2} \\ &= (r_1 r_2)^N(1+N)^{m_1+m_2} \\ &= \mathsf{Enc}(pk, m_1 + m_2), \end{aligned}$$

where $r_1 r_2$ and $m_1 + m_2$ are computed modulo N. (Technically, to ensure that $C := \mathsf{Enc}(pk, m_1) \cdot \mathsf{Enc}(pk, m_2)$ really is a *properly distributed* encryption of $m_1 + m_2$, we have to rerandomize C by multiplying with a fresh random value r^N.)

4.3 Applications to smart metering

Setting. In a smart metering system, we can think of securely aggregating measurements before transmitting them to the energy provider, in order to ensure (a certain degree of) confidentiality of the customer's data. Concretely, we can aggregate measurements 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.

A concrete protocol. [16] explain how the secure aggregation of measurements across several customers can be performed using an additively homomorphic encryption scheme such as Paillier's scheme. Concretely, the idea is as follows:

1. 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.

2. All customers engage in an efficient multiparty protocol to compute an encryption of the aggregation $\sum_i m_i$ of measurements under the energy provider's public key.

$$C := \mathsf{Enc}(pk, \sum_i m_i).$$

(Note that this does not involve point-to-point communication among the customers, but only a link from each customer to the energy provider.)

3. In the end, the energy provider decrypts C and (only) learns the aggregation of measurements, while no customer learns anything (on top of his or her own measurement of course).

How to go further. This approach of secure aggregation demonstrates the applicability of (limited) homomorphic encryption to the smart metering setting. In particular, [16] show how cryptography can be used to simultaneously achieve seemingly contradictory requirements (the energy provider's desire to gather information vs. the customer's privacy). Our goal is to extend these ideas for the use in a practical smart metering system.

For instance, as outlined in Section 3, an additional requirement present in a smart metering system is the integrity (i.e., authenticity) of measurements. Such an authenticity requirement can be fulfilled by digitally *signing* the measurements. However, *signed* measurements can no longer be easily accumulated (e.g., inside a Paillier encryption). It is an interesting and unique challenge to combine such authentication methods with aggregation techniques. (More specifically, we would want to aggregate signed pieces of data, such that an aggregated signatures authenticates the accumulated data.)

4.4 More examples

Secure aggregation of traffic data. As another example of the use of (limited) homomorphic encryption, consider the secure aggregation of traffic data. We could imagine a number of traffic sensors that measure the number of passing cars and periodically transmit measurements to a base station. Transmitting those measurements in plain and unencrypted could result in a loss of anonymity: it could become possible to track individual cars. Only encrypting measurements and sending them to the base station would still allow that base station to track individual cars. However, suppose now that encrypted measurements could be *aggregated*, in the sense that each sensor

- first receives an encryption of the so far aggregated measurements $\mathsf{Enc}(pk, \sum_{j=1}^{i-1} m_j)$ of the previous station,
- then homomorphically adds its own (encrypted) measurement $\mathsf{Enc}(pk, m_i)$ to that previous measurement,
- and sends the encrypted accumulated measurement $\mathsf{Enc}(pk, \sum_{j=1}^{i} m_i)$ to the next station.

In this scenario, a base station would only receive accumulated traffic information. Such accumulated information can still be helpful, e.g., to detect and potentially prevent traffic jams, but protects the privacy of individuals. In fact, we can even have a tradeoff between privacy and monitoring accuracy by adjusting the degree of accumulation. Thus, we have a system whose properties can be adjusted by fine-tuning parameters, similar to systems in algorithm engineering. Depending on the desired concrete application parameters, as well as security and efficiency goals, we can hope to find an optimal point in this continuum for a given specific application.

In this traffic analysis setting, a certain homomorphic property is required from the used encryption scheme, since it must be possible to aggregate natural numbers. However, very efficient encryption schemes—such as Paillier's encryption scheme—with such a limited accumulation property are well-known to exist. In particular, while fully homomorphic encryption would lead to an impractical solution, a practical solution can be found by using the specific structure of the problem.

Secure storage. Similarly, when merely large *storage* capacities in the cloud are required, again efficient solutions exist. For instance, consider a scenario in which a large medical database that includes individual patient records is to be outsourced into the cloud. Encrypting this data alone might not be sufficient to protect the privacy of individual patients. At the very least, a potentially curious server in the cloud might learn which (encrypted) records are accessed more often.

However, in this setting, cryptographic technologies such as *private information retrieval (PIR)* can be employed. PIR techniques allow for a comparatively efficient access to encrypted stored data, while the actual server on which that data is stored learns essentially only *that* an access took place (but not *which* part of the data was accessed).

4.5 Supporting FHE by other tools

Orthogonally, we can hope to use (fully) homomorphic encryption techniques in an efficient way when they are supported by additional cryptographic tools. For instance, imagine a small tamper-proof hardware device that performs arbitrary computations. Of course, one has to be careful in making such assumptions, since

(a) it takes a significant effort in hardware design to protect such a device against physical attacks, and

(b) since the device is small, we cannot assume that it is computationally very powerful.

Such hardware tokens can be used to bootstrap a very general class of secure computations. In particular, hardware tokens alone enable arbitrary secure two-party computations. (In a secure two-party computation, both parties get an input x_1, resp. x_2, and eventually receive an output $f(x_1, x_2)$, where the function f is agreed upon. It should be stressed that both parties do *not* learn anything about the other party's input, beyond $f(x_1, x_2)$ of course.) Many real-life protocol tasks (e.g., negotiations for a price) can be expressed as such a secure two-party computation.

However, constructing tamper-proof devices requires an expensive dedicated hardware design. In particular, if we have k different devices for the use in different contexts, we will have to design and protect k different pieces of hardware. Fully homomorphic encryption can come to our aid here: instead of designing k different hardware devices, we only design one universal *decryption* device. Such a device contains the secret key that is necessary to decrypt (fully homomorphic) encryptions. We can now build a larger device that first of all encrypts its inputs x_1 and x_2 (this can be done using a public encryption key), and then homomorphically computes an encryption of $f(x_1, x_2)$ from the encrypted x_i. Finally, the result is decrypted and output. Observe that only the initial encryption and final decryption steps actually have to be protected; the actual computation takes place on encrypted data and thus could even be performed publicly.

Along these lines, fully homomorphic encryption allows to generically construct arbitrary tamper-proof hardware from one single and very specific piece of hardware for encryption and decryption. In particular, an expensive hardware protection process has only to be performed once and for all. Arbitrary tamper-proof hardware devices can be derived almost canonically.

Of course, we still have to ensure that our solution is reasonably efficient. In particular, when using fully homomorphic encryption, we still suffer from a considerable slowdown. However, hardware tokens are already only used for certain protocol-critical operations for which hardware support is required to ensure security. (One can think of using hardware tokens only to store and thus physically protect long-term secret keys.) Hence we can hope that the use of computationally very expensive techniques like fully homomorphic encryption is much more practical than, e.g., in a generic cloud computing setting.

4.6 A tradeoff

We believe that the preceding examples demonstrate that it is crucial to use "heavy" cryptographic techniques like fully homomorphic encryption with care, with a lot of fine-tuning for the actual application. Again we have a tradeoff between efficiency and security, where the degree to which a cryptographic tool like fully homomorphic encryption is used determines the characteristics of an implementation.

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 DSCI, *integrity* will be essential, meaning that critical computations cannot be manipulated from outside. For the protection of privacy, *confidentiality* will be essential, meaning that private data cannot flow to public ports. However, both cannot be guaranteed with classical techniques alone: classical approaches do not really give guarantees about the *behaviour* of software, but rather about its *origin*.

Fortunately, 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". Modern program analysis based on interprocedural dataflow analysis, abstract interpretation, or model checking has developed very powerful tools for discovering anomalies in software. Experimental security infrastructures based on these techniques have been developed in large European projects [5]. IBM developed a tool for IFC which can analyse large programs written in full Java [35]. 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. In the following, we will describe some of the new security techniques, as well as their application to smart meters. Note that these techniques have many other applications in virtualized infrastructures.

5.1 Proof Carrying Code

Proof carrying code is code for software components (typically mobile components), which comes with an (encoded) formal proof of some desireable property of the software. 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 or plug-in, 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, appropriate formal logics as well as efficient 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.

```
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 Java password checker

In the smart meter application, 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 – will disallow installation.

5.2 Information Flow Control

Proof carrying code can guarantee arbitrary functional or security related properties, but requires expensive proof preparation and nontrivial checkers. As an alternative, new techniques for language-based security can be applied to guarantee integrity and privacy. In particular, *information flow control* analyses the program source or byte code for security leaks. Data which are 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

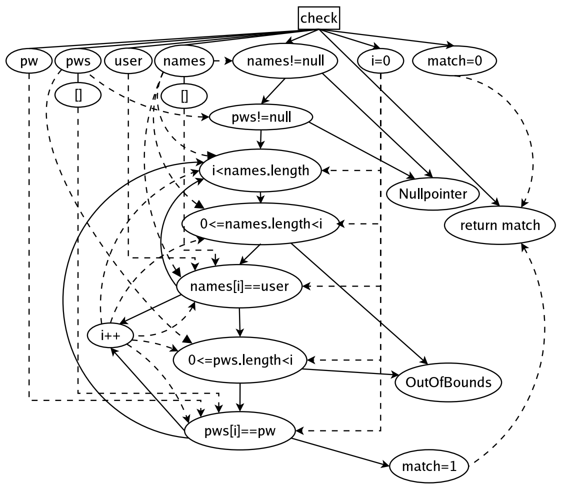


Figure 3: Program dependency graph for figure 2 (exceptions included)

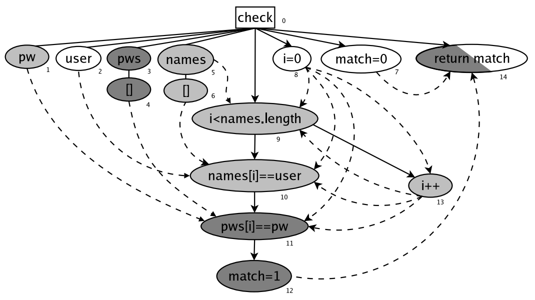


Figure 4: Program dependency graph for figure 2 (exceptions excluded) with computed security levels (white=public, grey=confidential, dark=secret). The program contains a security leak showing up as a level conflict in the return node (upper right). Indeed, there is an information flow from the secret password table to the public output, which can be exploited.

(e.g. appliance switching commands) must not be manipulated from outside (e.g. by 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 multithreading 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, the new SPP "reliably secure software" integrates information flow control with modern program analysis and verification technology. Let us thus describe one such approach, as developed in the group of G. Snelting [22], in more detail.

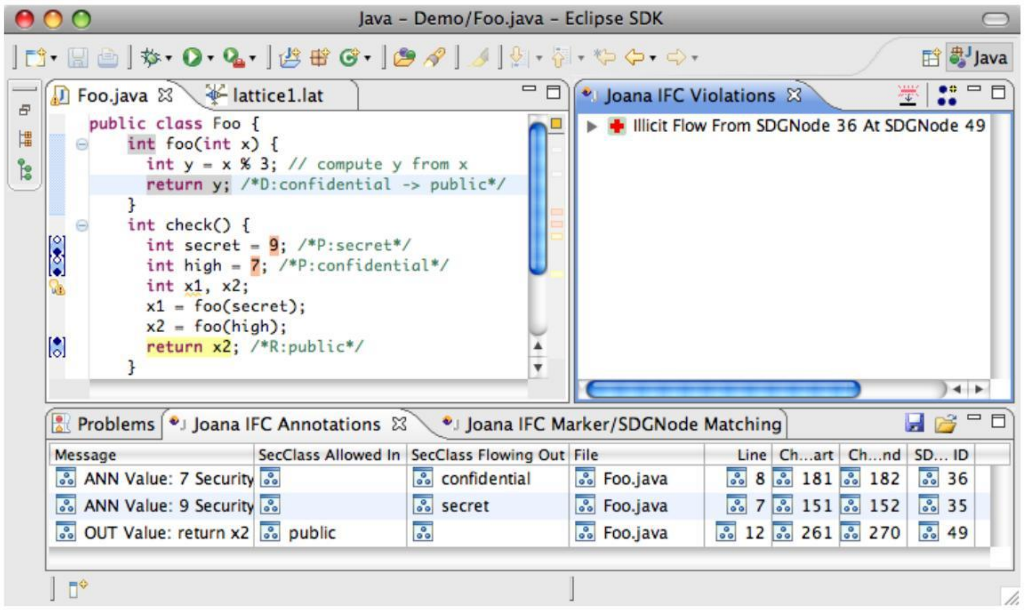
Security type systems, as used by Mobius, have been an important step and are quite efficient, but can be unprecise, resulting in false alarms. A more precise analysis must exploit flow-sensitive, object-sensitive, and context-sensitive information as computed by interprocedural dataflow analysis. The results of such an analysis can be encoded in form of a program dependency graph, as indicated in figure 3. 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 [38]) 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. Hundreds of papers have been written on the subject; today, two dependency graph implementations for full Java exist (one, the JOANA tool, developed in Snelting's group), as well as a commercial implementation for C/C++, called CodeSurfer.

For information flow control, input and output ports in the graph must be annotated with security levels. For the rest of the program resp. its dependency graph, security is checked by a fixpoint iteration which is based on the following fundamental equations:

$$S(x) \geq \begin{cases} P(x) \sqcup \bigsqcup_{y \in pred(x)} S(y), & \text{if } x \in dom(P) \\ \bigsqcup_{y \in pred(x)} S(y), & \text{otherwise} \end{cases}$$

$$R(x) \geq S(x) \quad \text{if } \ x \in dom(R)$$

where S is the security level of a graph node x, P the annotation of an input port, and R the annotation of an output port. For figure 2, the resulting security levels are shown in figure 4. The JOANA analysis can handle full Java bytecode and scales up to 50kLOC; it is implemented as an Eclipse plug-in (figure 5). Full details can be found in [22, 20, 21]. The analysis is currently adapted for mobile components in the scope of the above-mentioned SPP.



```
public class Foo {
    int foo(int x) {
      int y = x % 3; // compute y from x
      return y; /*D:confidential -> public*/
    }
    int check() {
      int secret = 9; /*P:secret*/
      int high = 7; /*P:confidential*/
      int x1, x2;
      x1 = foo(secret);
      x2 = foo(high);
      return x2; /*R:public*/
    }
```

```
Illicit Flow From SDGNode 36 At SDGNode 49
```

Message	SecClass Allowed In	SecClass Flowing Out	File	Line	Ch...art	Ch...nd	SD... ID
ANN Value: 7 Security		confidential	Foo.java	8	181	182	36
ANN Value: 9 Security		secret	Foo.java	7	151	152	35
OUT Value: return x2	public		Foo.java	12	261	270	49

Figure 5: Eclipse plugin for information flow control

For smart meters and many other software in virtualized infrastructures, it will be necessary to apply information flow control to (selected parts of) the source code. In particular, the analysis can guarantee that integrity of the trusted device cannot be broken by software attacks. This is in turn 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 pricavy protection by introducing appropriate security levels for secret, encrypted, aggregated, and public data; analysing the information flow for all such data in the smart meter and the cockpit, and by carefully introducing declassification [30] e.g. at aggregation points.

For software outside the smart home, a complete information flow analysis will not be possible due to the tremendous software size and the number of stakeholders. Still, information flow control can analyse critical software kernels, but must be combined with more traditional technology such as cryptography, certificates and mandatory access control. Static information flow control will also be extended by dynamic analysis and runtime verification, as described in the next chapter.

6 Deductive Program Verification

6.1 Deductive Verification for Ensuring Confidentiality and Integrity in Smart Meters

Various measures can be taken to ensure the confidentiality and integrity of software in smart meters. But for the smart meter to be trustworthy, in the end it is indispensable that the kernel software in the trusted device is functionally correct.

Even if other techniques (e.g., run-time checking or proof-carrying code) are used to ensure critical properties, certain functionality of the kernel must be verified in addition (e.g., it must be shown that the run-time checker is implemented correctly). One may use information-flow control to show that communication mechanisms are used in such a way that confidentiality is preserved. But one must still verify that these communication mechanisms are implemented correctly and do not allow information leaks.

Thus, validating the functional correctness of the trusted device's system kernel is central to ensuring the integrity of the smart metering system. And since bugs in the kernel could be exploited for system-wide attacks against a critical infrastructure, it is justified to use heavy-weight formal methods – such as deductive program verification – to ensure the kernel's correctness.

Formal methods are also needed because smart metering technology combines two trade-offs in a complex way: confidentiality vs. intelligent control, and integrity vs. adaptability and openness. This entails that properties need to be ensured that balance these trade-offs and are accordingly complex and difficult to formulate. The integration with the physical world adds further complexity.

There are different possibilities for who performs the verification of different system parts. In particular, we applications and device drivers running on top of the trusted kernel. A certification agency may be involved in different roles. It may validate the softare itself, it may check a verication performed by the system developer, it may certify tools used for verification, or it may provide (formalisations of) properties, and/or tools that allow the user to check evidence provided by the developer (proof-carrying code).

6.2 Deductive Verification of System Code

Overview. The field of deductive program verification, i.e., formal reasoning about the behaviour of programs, is old. The idea of applying deduction to programs goes back at least to the work of Scott, Plotkin, and Milner in the late 1960s. Recent years have brought tremendous advances in both scope and practicality, however. Today, program verification is applied to real-world software. For example, security-critical system software is verified in the Verisoft XT project (see, e.g., the paper by [9] on deductive verification in Verisoft XT) and the L4.verified project (see the overview paper by [27]).

As an example for a successful method for deductive verification of system code, we below describe the approach used in the Verisoft XT project. While Verisoft XT did not lead to a full verification of a mikro kernel (mostly due to a lack in time and man-power), it was clearly demonstrated that a complete verification is feasible. The kernel considered in Verisoft XT, SYSGO's PikeOS, may very well serve as the basis for implementing a trusted device kernel for an advanced smart meter implementation.

Verisoft XT: Verifying the PikeOS Micro Kernel In the first phase of the Verisoft project it has been shown that pervasive formal verification of an academic operating system including its execution environment, like the underlying hardware and the compiler, is feasible.

In the subproject Avionics of the successor project Verisoft XT, this knowledge was applied and refined to the verification of a real world implementation of a microkernel used in industrial embedded systems, namely PikeOS from SYSGO AG which operates in safety-critical environments. One goal of the Verisoft XT subproject Avionics was to prove functional properties of the source code of the microkernel using Microsoft's verification tool VCC [10].

PikeOS (see `http://www.pikeos.com/`) consists of a microkernel acting as paravirtualizing hypervisor and a system software component. The PikeOS *kernel* is particularly tailored to the context of embedded systems, featuring real-time functionality and orthogonal partitioning of resources such as processor time, user address space memory and kernel resources.

PikeOS could easily be adapted to the requirements of a smart metering system. It would, of course, have to be extended by additional functionality for this particular application, such as encrypted communication, switching appliances etc.

The Verifying Compiler Approach. It is widely recognized that interaction is indispensable in deductive verification of real-world code. Verification engineers have to guide the proof search and provide information reflecting their insight into the workings of the program. Lately we have seen a shift towards a paradigm, called verifying compilers [25], where the required information is provided in form of program annotations instead of interactively during proof construction. This has some interesting consequences upon the verification process and the way annotations are used to specify programs as the lines between requirement specification and information required for proof construction and proof search guidance get blurred.

Also, verifying compilers allow for new ways of coping with programming language semantics. Instead of directly axiomatizing the complex semantics of the high-level programming language, verification is done at the level of an intermediate language with clear and simple semantics. A prominent example is Microsoft's BoogiePL [13], which is used in Spec# and VCC among other tools. Annotated code in the intermediate language is typically obtained from annotated source code by using compiler technology. Despite additional problems – the transformation from annotated source code to intermediate code, for example, obfuscates the verification problem and makes it harder to map verification results back to the source code level –, the use of an intermediate language offers substantial advantages. It facilitates adaptation to other programming languages, but foremost it allows a separation of concerns, namely the semantics of the source code programming language on the one hand and the genuine verification problem on the other. Also, intermediate languages usually have constructs, such as a non-deterministic choice operator, that are difficult to include in a real programming language but are very useful for formal specification and verification.

Tools following the verifying compiler paradigm include Spec# [4], VCC [34], and Caduceus [15]. They are all based on powerful fully-automatic provers and decision procedures, and they support real-world programming languages such as C and C#. VCC was used in the Verisoft XT project.

Verification in VCC is modular, both with respect to threads and functions. Functions are equipped with contracts in form of pre- and post-conditions, giving all necessary conditions to call the function and the guarantees on the state, when the function returns. Callers are then verified with respect to the contracts, not bodies, of the called functions. The program is verified as if it were executed by a single thread but, to handle concurrency, predicates describing knowledge about the state are weakened at possible points of interleavings to simulate the effects of other threads.

Specifying a microkernel with a simulation relation. As explained above, properties of a smart metering system that need to be verified to ensure integrity and confidentiality are rather complex and difficult to formulate.

The standard approach to specifying the required properties on an abstract level is a simulation theorem. The proof is conducted by inductively showing that each step of the specification, which includes an abstract model of the smart meter's trusted device, is realized by a certain number of steps in the implementation.

For example, a simulation theorem has been developed and proven in the first Verisoft project [36]. While a real-world micro kernel differs from the "academic" system used in Verisoft I (e.g., full C semantics, interruptible kernel, shared memory, real-world architecture), the principal approach to show its correctness remains the same: formally verifying a simulation theorem between an abstract specification and the concrete implementation of the system.

For a simulation proof we need to look at the system at different layers of abstraction. In our case there are three of them. The first and most abstract one is *cvm* – the specification model. It consists of an *abstract kernel* that specifies the user-visible parts of the implementation and hides hardware functionality. The other part consists of the additional (possibly untrusted) processes running on the system. We interpret these processes as separate *virtual machines* that communicate with each other only via defined channels (e.g. shared memory, IPC). The *concrete kernel* layer represents the C and assembly implementation which precisely describes the functionality of most parts of the kernel, given one has assigned an unambiguous semantics to C by fixing a compiler and an architecture. Finally, the *architecture* layer models the physical hardware on which assembly code, compiled C code, and the additional processes are executed. Formally we can model these layers as follows

- *cvm* – the abstract model consisting of:
  - $cvm.vm(i)$ – the virtual machine of the i-th process, consisting of the a CPU context $vm(i).cpu$ and a virtual memory portion $vm(i).m$ of some adjustable size.
  - $cvm.c(i)$ – the C configuration of the abstract kernel thread i, comprising components like program code or a local memory stack and sharing global memory with the other threads. Note that $c(i)$ only becomes active when $vm(i)$ enters the kernel (e.g., via a system call).
- $k(i)$ – the C configuration of the concrete kernel thread which implements $c(i)$, including additional data structures not visible from outside and assembly code.

- h – the model of the underlying hardware *architecture*, basically comprising the CPU context $h.cpu$ and physical memory $h.m$.

One can then define relations connecting the different layers. For instance, we define a *B-relation* [17] that relates specification and implementation of the additional processes. It states that the context of the active process agrees with the CPU registers and all other processes are encoded in dedicated data structures of the kernel. For the virtual machines' memory it demands that memory contents are equal to those of the corresponding regions on the physical machine. There is also an *abstraction relation* between abstract and concrete kernel as well as a *compiler consistency relation* between C code and compiler-generated assembly code, that guarantee that the concrete kernel program is correctly executed on the underlying hardware.

Formally, one combines these relations into an overall relation $\mathit{cvm\text{-}sim}(\mathit{cvm}, k, h)$ stating that the cvm model is simulated by the concrete kernel k and the hardware state h. In addition there are implementation invariants $\mathit{impl\text{-}inv}(\mathit{cvm}, k, h)$ for specific layers, which specify that the contained components and data structures remain well-defined.

For any n execution steps in the cvm model a trace of m hardware steps can be found that simulates the cvm execution, such that all three layers are consistent to each other.

With transitions on the cvm model and the hardware defined by step functions δ_{cvm} and δ_h, the overall simulation theorem between cvm, concrete kernel and architecture layer can be stated as follows. Assuming validity and induction start preconditions on the initial configurations cvm^0 and h^0 we have:

$$\forall\, n\ \exists\, m\ \exists\, k \big(\mathit{impl\text{-}inv}(\delta^n_{\mathit{cvm}}(\mathit{cvm}^0), k, \delta^m_h(h^0)) \wedge \\ \mathit{cvm\text{-}sim}(\delta^n_{\mathit{cvm}}(\mathit{cvm}^0), k, \delta^m_h(h^0))\big)$$

6.3 Deductive Verification of Information-flow Properties

As said above, tremendous progress has been achieved in formal verification of functional properties of software. At the same time seminal papers have been published showing that it is in principle possible 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 [12] 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. This notion is employed and made explicit in the information-flow analysis.

In a smart-metering system, more complex properties such as controlled information release need to be assured. Verifying such properties is a current hot research topic. We will carry out related research in the DeduSec project within the DFG Priority Programme 1496 "Reliably Secure Software Systems – RS3". In this project, 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 by us) [8].

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 (e.g., [6, 12, 37]); or it can be translated into dependence properties, which in turn can be formalized in program logics logic (this has been investigated for a simple imperative language [2, 1], for a simple object-oriented language [3], and for sequential Java [19]). 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* [12, 11, 6].

We also plan to adapt the concept of *ownership* 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. [28]). For information-flow properties, ownership has to be adapted so that one can specify that data structures are not *read* in an unintended way.

6.4 Further Challenges

Adequacy of Requirements and Certification As explained in this report, enforcing confidentiality and integrity of a smart metering system involves a varity of measures. While the deductive methods described above mostly apply to the implementation at code-level, the verified properties must be related to higher-level requirements (e.g., policies). Relating these levels to each other is a scientific challenge that still demands research.

Also, it is important for certification, that not only are the verified properties adequate and validation and analysis techniques are correctly applied but that this adequacy and correctness can be checked and validated by third parties. This requires further research and extensions of existing verification methods.

Verification of Evolving Software. For evolving software, analyses of properties have to be repeated. This fact has not been addressed in current software verification and certification approaches, which are design-once-change-never oriented. Most quality assurance methods are challenged by adaptability. How to adapt and "repair" verification proofs and formal models after an adaptation is an unsolved problem, and verifying self-adaptive systems is a great challenge.

7 Data Usage Control with Runtime Verification and Dynamic Data Flow Analysis

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.

This problem spans three dimensions. The first dimension is usage control proper, as found in digital rights management systems: given a specific data item, how can the usage – events including printing, saving, copying, etc. – of this data be controlled. Typical solutions to this problem include, among other things, runtime verification (the scientific roots of which are temporal logics and automata theory), and complex event processing (the scientific roots of which are active data base technology and event-condition-action rules). The second problem dimension is data flow analyis across different representations. Usage control mechanisms, as mentioned above, are fundamentally bound to the notion of events and usually do not consider data at all. As such, the events in question are usually parameterized with one concrete representation of a sensitive data item. However, usage control policies are usually meant to be concerned not with one but rather with *all* representations of the data. For instance, if the customer's master data should not be copied, this requirement applies to both some textual representation and the pixel representation on a screen. Similarly, daily energy consumption comes in the form of a number of raw measurements as well as in the form of some graph on a screen. If the raw data must not be copied, then this means that a screenshot of the graphical representation must not be taken as well. The scientific core of this problem is, on the one hand, data flow and information flow analysis within one layer of abstraction, e.g., within .NET CIL or within some RTL language. On the other hand, data flows in-between these levels of abstraction must be monitored, which is a rather new and open problem. Finally, the third problem dimension is distribution: Data flows must not only be detected (second problem dimension) and controlled (first problem dimension) within one of the IT systems represented by boxes in Figure 1, but also in-between different systems and governance domains. In other words, different representations may exist on different machines, and all of them must be controlled.

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 managament 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 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 [14].

7.1 Runtime Verification

Roughly speaking, runtime verification denotes a set of techniques that implement 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.

There is a variety of algorithms for performing runtime verification with a variety of optimality results concerning, among other things, the possibility to decide on truth or falsity of a formula at the earliest possible moment in time, the number of states that need to be stored, etc. [29].

For controlling data usage, a simple temporal logic with abstractions for limited cardinality constraints is the Obligation Specification Language, or OSL [23]. As we will explain below, traces are sequences of sets of events. Then, given an OSL formula φ and a trace (prefix) t, runtime verification decides at runtime, for each moment in time n, whether or not φ 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.

7.1.1 System Model

We introduce the syntax and semantics of OSL. We formalize both in Z, a formal language based on typed set theory and first-order logic with equality. We have chosen Z because of its rich notation, which we explain as it is encountered. We have also given a more user-friendly syntax to OSL [24], which we do not present here for brevity's sake. The current version of OSL supports all usage control requirements identified above, except environment conditions.

The semantics of our language is defined over traces with discrete time steps. At each time step, a set of events can occur. An event corresponds to the execution of an action and we use these two terms interchangeably.

Each *Event* has a name and parameters, specifying additional details about the event. For example, a usage event can indicate on which data item it is performed or by which device. An example of an event is $(snd, \{(obj, o), (rcv, r)\})$, where *snd* is the event name and the parameter with name *obj* has value o while the value of *rcv* is r—intuitively, the object o is sent to receiver r.

Each event belongs to an event *class*. Possible event classes include *usage* and *other*, the latter standing for all non-usage events, e.g., payments or notifications. This distinction enables us to prohibit all usages on a data item while still allowing other events such as payments.

7.1.2 Syntax

An OSL policy consists of a set of event declarations and a set of obligational formulae. Each obligational formula consists of the data consumer's name and a logical expression.

Φ defines the syntax of the logical expressions contained in obligational formulae, as shown in Figure 6. For brevity's sake, we omit the formal definition of the set of events, *Event*, here—we may simply assume this set to be given. $E_{fst}(e)$ refers to the start of an event e and $E_{all}(e)$ to ongoing events.

We define an additional restriction on the policy syntax (omitted here): we demand that all events that are mentioned in a policy are compliant with the event declaration, i.e., they may only contain parameters that are declared and corresponding values. Fewer parameters are allowed in a policy, because of the implicit universal quantification over unspecified parameters.

7.1.3 Informal Semantics

We informally describe the semantics of OSL's operators here; a formal definition is provided elsewhere [23]. They are classified into propositional operators, temporal operators, cardinality operators, and permit operators, the latter of which we do not discuss here.

Propositional Operators The operators $\underline{not}$, $\underline{and}$, $\underline{or}$, and $\underline{implies}$ have the same semantics as their propositional counterparts $\neg, \wedge, \vee$, and $\Rightarrow$.

Temporal Operators The $\underline{until}$ operator corresponds to the *weak until* operator from LTL [33]. We use the weak version of the until operator because it is better suited for expressing usage control requirements (cf. §**??**). We generalize the *next* operator of LTL to $\underline{after}$, which takes a natural number n as input and refers to the time after n time steps. With $\underline{after}$, we can express concepts like $\underline{during}$ (something must hold constantly during a given time interval) and $\underline{within}$ (something must hold at least once during a given time interval).

Cardinality Operators Cardinality operators restrict the number of occurrences of a specific event or the accumulated duration of an event. The $\underline{repuntil}$ operator limits the maximum number of times an event may occur until another event occurs. For example,

$$\underline{repuntil}(15, E_{fst}(snd, \{(obj, sd), (rcv, ep)\}),\\ E_{fst}((chck, \varnothing)))$$

states that sensor data *sd* is sent at most 15 times to the energy provider *ep* before a self check event *chck* must take place. With $\underline{repuntil}$, we can also define $\underline{repmax}$, which is syntactic sugar for defining the maximum number of times an event may occur in the unlimited future.

A policy is satisfied by a trace iff all obligations specified in the policy are satisfied by the trace. The definition of obligation satisfaction builds on the above semantics but requires a system model that includes activations of obligations. Such a complete system model is presented in [24].

7.2 Dynamic data flow analysis

In the following, we assume a reserved parameter, *obj*, indicating which object the event is related to and a reserved value for that object *nil*, used to indicate no object. In the case of events that need more than one object parameter (like copy, which requires a source and a destination), we assume the presence of a single *obj* parameter only; other parameters will be defined using different names. For instance, the syntax for a send command will be similar to $send(\{(obj, obj1), (dst, obj2)\})$.

7.2.1 Data Items and Data Containers

To the end of data flow analysis, we need to introduce the distinction between *data* items and *containers* for data items. Roughly, the idea is that in order to control all copies of a data item, we keep track of all its representations, or containers. Containers are the different representations of data, including files, database records, network packets, memory regions, etc. This leads to the distinction between two classes

$$\begin{aligned}\Phi ::= {} & \underline{true} \mid \underline{false} \mid E_{fst}\langle\langle Event\rangle\rangle \mid E_{all}\langle\langle Event\rangle\rangle \mid \underline{not}\langle\langle\Phi\rangle\rangle \mid \underline{and}\langle\langle\Phi\times\Phi\rangle\rangle \mid \underline{or}\langle\langle\Phi\times\Phi\rangle\rangle \mid \\ & \underline{implies}\langle\langle\Phi\times\Phi\rangle\rangle \mid \underline{until}\langle\langle\Phi\times\Phi\rangle\rangle \mid \underline{always}\langle\langle\Phi\rangle\rangle \mid \underline{after}\langle\langle\mathbb{N}\times\Phi\rangle\rangle \mid \underline{within}\langle\langle\mathbb{N}\times\Phi\rangle\rangle \mid \\ & \underline{during}\langle\langle\mathbb{N}\times\Phi\rangle\rangle \mid \underline{repmax}\langle\langle\mathbb{N}\times\Phi\rangle\rangle \mid \underline{repuntil}\langle\langle\mathbb{N}\times\Phi\times\Phi\rangle\rangle\end{aligned}$$

Figure 6: Syntax of OSL

of events, according to the type of the *obj* parameter: events of class *dataUsage* define actions on data objects, while events of class *containerUsage* refer to a single container.

Within the system, only events of class containerUsage can happen, because each monitored event in a trace is related to a specific representation of the data. DataUsage events are used only in the definition of policies, where it is possible to define a rule abstracting from the specific representation of the information. Accordingly, we define two subsets of events, *CEvent* and *DEvent*, respectively for events of class *dataUsage* and *containerUsage*.

We demand that all events of class *usage* have an object parameter. This parameter indicates the object the event is referred to. So, as we discussed before, *ParamName obj* for events of class *dataUsage* has to be mapped to data, as well as for events of class *containerUsage* it has to be mapped to a container.

Of course, the system has to satisfy some additional sanity constraints that we omit for brevity's sake.

7.2.2 Data State

In order to integrate information flow detection capabilities into the semantic model of OSL, we need to add also functions for modeling the relationship between containers and data.

The same data can be stored in multiple containers. Multiple data items can be stored in a single container. We model this *n-to-n* relation with two functions, one from *Data* to a set of *Container*, and another one from a *Container* to a set of *Data*. Although one can be derived from the other, we use two functions for simplicity's sake.

We also have to define an *InitialCont* function, a bijective mapping between data and containers that represents the initial container that stores a data item as soon as it starts to be monitored by the system.

Moreover, we introduce the *Alias* function to model the relation between connected containers. By connected, we mean that a content update of the first implies a content update of the other ones. This happens when, for instance, multiple containers are mapped to (totally or partially) overlapping memory areas. In this case, writing data in one of them implies writing in the other ones. Last but not least, we define the *Naming* function from a set of names (a subset of the set of parameter values, *ParamValues*) to Container. As discussed before, this is useful to model renaming activities.

$$\begin{aligned}&Name : \mathbb{P}\, ParamValue\\&Alias : Container \nrightarrow \mathbb{P}\, Container\\&Naming : Name \nrightarrow Container\end{aligned}$$

Now we are ready to define a *state* (*IFState*) of the information flow model as the triple $Storage \times Alias \times Naming$. The transition relation among states is of course dependent on the system that is modeled. We define *IFR* as $IFState \times Event \nrightarrow IFState$. According to the semantic model of OSL, at each time step, a trace consists of a set of events rather than a single one. For this reason we need to define a state transition relation *IFRSet* of type $IFState \times \mathbb{P}\, Event \nrightarrow IFState$. If all the events in the set are independent, then this is equivalent to the union of *IFR* applied to the set. But this being not always the case, we must consider transitions caused by a set of events.

We need to consider a particular state Σ_i where the storage function contains only an empty mapping for the reserved object *nil* and the alias function is empty. W.l.o.g we can assume this to be the initial state of the system.

$$\begin{aligned}&IFState : Storage \times Alias \times Naming\\&IFR : IFState \times Event \nrightarrow IFState\\&IFRSet : IFState \times \mathbb{P}\, Event \nrightarrow IFState\\&\Sigma_i == ((nil, \varnothing), \varnothing, \varnothing)\end{aligned}$$

7.2.3 Syntax and State based formulae

In order to monitor data flows, we keep track of the data state: which containers contain which data. Our extension of OSL will now be evaluated not only over traces of events, but also over states of the data flow model.

To define *state-based formulae* we add an operator $\underline{state}\langle\langle\Phi_s\rangle\rangle$ to Φ on the grounds of a new set of *state-based operators* Φ_s. In order to express constraints on data instead of containers, we introduce three new operators, $\underline{denyC}$, $\underline{denyD}$ and $\underline{limit}$.

$$\begin{aligned}\Phi_s ::= {} & \underline{denyC}\langle\langle Data \times \mathbb{P}\, Container\rangle\rangle \mid \\ & \underline{limit}\langle\langle Data \times \mathbb{P}\, Container\rangle\rangle \mid \\ & \underline{denyD}\langle\langle Data \times Data\rangle\rangle\end{aligned}$$

7.2.4 Informal Semantics

We can concentrate on the new construct $\underline{state}()$ and on the set Φ_s. The $\underline{state}()$ operator is needed to syntactically merge the new state-formulae with the original system while keeping the two models separate: pure state formulae appear as argument of a $\underline{state}()$ function.

Intuitively, $\underline{denyC}(d, C)$ forbids the presence of data d in one of the containers in set C. This operator is useful to express constraints, like for instance "profile s must not be distributed over the network", which becomes $\underline{denyC}(s, \{c_{net}\})$. The rule $\underline{denyD}(d_1, d_2)$ claims that data d_1 and data d_2 cannot be combined, which means they can never be in the same container.

$\underline{limit}(d, C)$ is the dual of $\underline{denyC}$: it expresses the constraint that data d can only be in containers of set C. If AC is the set of all possible containers of the system, then $\underline{denyC}(d, C)$ is equivalent to $\underline{limit}(d, AC \setminus C)$. This can be used to express concepts like "data d must be deleted", $\underline{limit}(d, \varnothing)$, which is useful for forensic analyses.

7.2.5 Implementation

We have implemented generic technology to perform this data flow tracking. In the SPP 1496, Reliably Secure Software System, we are currently working on a general schema for connecting different layers of abstraction (which, to iterate, has not been done in the case of the smart metering system yet). It is noteworthy here that the static information flow detection technologies from Sections 5 and 6 are likely to, at one single layer, substitute dynamic detection techniques for the same layer. This is particularly appealing if the static techniques prove to be very precise, which appears to be the case for Java bytecode, for instance, when not too much dynamic binding takes place.

7.2.6 Application to Smart Metering

With the help of OSL, augmented by constructs to speak of a system's data state, it is possible to specify policies that allow or disallow the flow of information within a distributed system, even when the boundaries of internal components are crossed. This is formally 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.

More concretely, we would need to deploy several of these monitors at different locations in the system. One monitor tracks the data flow within the smart meter itself (that is, the trusted device). In reality, this will not be one monitor but rather a set of monitors that monitor data flows at and in-between the different levels of abstraction within the trusted device, including the operating system and application layers. Another monitor is required for the cockpit. This is a full-fledged PC, so the monitor again consists of a set of monitors that track data flow at and in-between the different layers of abstraction of this PC, including the operating system, window manager, data bases, and applications like web browsers or email clients. At this stage, the granularity of data to be monitored also changes; we are more likely to speak of user profiles than of single measurements at this stage. In case the cockpit communicates with third party software, either Web 2.0 media, or billing or CRM software, then these systems need to be monitored in an identical way; and this process continues when considering the fact that data may be forwarded to call centers.

For all of these different systems, we need to either write or generate OSL policies to configure the generic runtime monitors that implement usage control and data flow detection. As mentioned above, 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 report, 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, allows 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 information flow in smart meters against predefined privacy policies expressed as dynamic (temporal) properties in case static IFC is not possible or too unprecise, or system boundaries need to be crossed.

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 (SCADA being abundant in critical infrastructures); such as the recent Stuxnet attacks:

- Stuxnet used stolen certification keys. This highlights the approach of DSCI and RS3, namely that we need to analyse the true semantics of a program and not just certify its origin. It is not clear whether today's language-based security techniques can analyse the full Stuxnet code, but program analysis and IFC are becoming more powerful every year.
- 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 bug-finding or IFC, which help to discover such anomalies.
- Verification, while expensive, can today formally verify realistic systems such as SCADA security cores or even operating systems.
- 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 DSCI security approaches. But techniques as proposed in the current article will certainly make attacks much more difficult, not just on smart meters, but on general SCADA systems, and on critical infrastructures as a whole.

We plan to actually develop and apply the techniques in the scope of the DSCI cluster initiative. If funding is agreed, work will start in 2012. We plan to engineer available methods for usage in E-Energy, E-Traffic, and Cloud systems; as well as to develop new approaches to security. Several demonstrators will be used to evaluate the DSCI approach, such as the "KIT Smart Home" and the "KIT Federated Cloud". The Smart Meter example will be the first realistic case study for our new approaches to dependable software in critical infrastructures.

References

[1] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 91–102. ACM, 2006.

[2] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In Roberto Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *LNCS*, pages 100–115. Springer, 2004.

[3] Torben Amtoft and Anindya Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Sci. Comput. Program.*, 64(1):3–28, 2007.

[4] Mike Barnett, Rustan Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS), International Workshop, 2004, Marseille, France, Revised Selected Papers*, LNCS 3362, pages 49–69. Springer, January 2005.

[5] Gilles Barthe, Lennart Beringer, Pierre Crégut, Benjamin Grégoire, Martin Hofmann, Peter Müller, Erik Poll, Germán Puebla, Ian Stark, and Eric Vétillard. Mobius: Mobility, ubiquity, security. objectives and progress report. In *TGC 2006: Proceedings of the second symposium on Trustworthy Global Computing*, LNCS. Springer-Verlag, 2006.

[6] 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.

[7] B. Beckert, T. Dreier, A. Grunwald, T. Leibfried, J. Müller-Quade, R. Reussner, P. Sanders, H. Schmeck, G. Snelting, S. Tai, W. Tichy, P. Vortisch, D. Wagner, and M. Zitterbart. Dependable software for critical infrastructures: Computing, energy, mobility. Project proposal, Karlsruher Institut für Technologie, 2010.

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

[9] Bernhard Beckert and Michał Moskal. Deductive verification of system software in the Verisoft XT project. *KI*, 2009. Online first version available at SpringerLink.

[10] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proc. TPHOLs 2009*, LNCS 5674, pages 23–42. Springer, 2009. Invited paper.

[11] Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Roberto Gorrieri, editor, *Workshop on Issues in the Theory of Security, WITS*. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.

[12] Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Proc. 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer, 2005.

[13] Rob DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[14] William Enck, Peter Gilbert, Byung-Gon Chun, Landon Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. 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.

[15] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering*, LNCS 3308, pages 15–29. Springer, 2004.

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

[17] Mauro Gargano, Mark A. Hillebrand, Dirk Leinenbach, and Wolfgang J. Paul. On the correctness of operating system kernels. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *LNCS*, pages 1–16. Springer, 2005.

[18] 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.

[19] Christian Haack, Erik Poll, and Aleksy Schubert. Explicit information flow properties in JML. In *3rd Benelux Workshop on Information and System Security (WISSec)*, November 2008.

[20] Christian Hammer. *Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs*. PhD thesis, Universität Karlsruhe (TH), Fak. f. Informatik, July 2009. ISBN 978-3-86644-398-3.

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

[22] 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.

[23] 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.

[24] Manuel Hilty, Alexander Pretschner, Thomas Walter, and Christian Schaefer. A system model and an obligation lanugage for distributed usage control. Technical Report I-ST-20, DoCoMo Euro-Labs, 2006.

[25] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

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

[27] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

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

[29] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, may/june 2009.

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

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

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

[33] Amir Pnueli. The temporal semantics of concurrent programs. In *Proc. International Sympoisum on Semantics of Concurrent Computation*, pages 1–20, 1979.

[34] Wolfram Schulte, Xia Songtao, Jan Smans, and Frank Piessens. A glimpse of a verifying C compiler. In *Proceedings, C/C++ Verification Workshop*, 2007.

[35] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 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.

[36] Alexandra Tsyban. *Formal Verification of a Framework for Microkernel Programmers.* PhD thesis, Dept. Computer Science, Saarland Univ., 2009. `http://www-wjp.cs.uni-sb.de/publikationen/Tsy09.pdf`.

[37] Martijn Warnier. *Language Based Security for Java and JML.* PhD thesis, Radboud University, Nijmegen, The Netherlands, 2006.

[38] Daniel Wasserrab. Backing up slicing: Verifying the interprocedural two-phase horwitz-reps-binkley slicer. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *The Archive of Formal Proofs.* `http://afp.sf.net/entries/HRB-Slicing.shtml`, November 2009. Formal proof development.