Int. J. Inf. Secur.
DOI 10.1007/s10207-016-0355-8

REGULAR CONTRIBUTION

Privacy-preserving smart metering revisited

Alfredo Rial[1] · **George Danezis**[2] · **Markulf Kohlweiss**[3]

Abstract Privacy-preserving billing protocols are useful in settings where a meter measures user consumption of some service, such as smart metering of utility consumption, pay-as-you-drive insurance and electronic toll collection. In such settings, service providers apply fine-grained tariff policies that require meters to provide a detailed account of user consumption. The protocols allow the user to pay to the service provider without revealing the user's consumption measurements. Our contribution is twofold. First, we propose a general model where a meter can output meter readings to multiple users, and where a user receives meter readings from multiple meters. Unlike previous schemes, our model accommodates a wider variety of smart metering applications. Second, we describe a protocol based on polynomial commitments that improves the efficiency of previous protocols for tariff policies that employ splines to compute the price due.

Keywords Universally composable security · Privacy · Billing · Smart meters · Polynomial commitments

✉ Alfredo Rial
alfredo.rial@uni.lu

George Danezis
g.danezis@ucl.ac.uk

Markulf Kohlweiss
markulf@microsoft.com

[1] University of Luxembourg, Esch-sur-Alzette, Luxembourg

[2] University College London, London, UK

[3] Microsoft Research, Cambridge, UK

1 Introduction

In privacy-preserving billing, a meter measures a user's consumption of some utility or service and service providers apply fine-grained tariff policies, i.e., policies that require detailed and frequent consumption measurements, in order to determine the bill.

A classical example is smart metering of electricity, water and gas [36]. In this setting, utility providers install smart meters in households in order to measure user consumption. Smart meters provide meter readings to the service provider. These readings are used by the service provider to calculate the bill under the tariff policy. The tariff policy may be complex, e.g., by applying a different rate depending on the time of consumption or on whether the consumption measurement reaches a threshold.

Other examples are electronic toll collection [24] and pay-as-you-drive car insurance [7]. In these cases, drivers install a meter in their cars that reports to the service provider which roads are used and when. Typical tariff policies apply different rates depending on the type of road (e.g., motorway, street), the time of the day (e.g., day or night) or even the speed of the vehicle.

In all the settings above, billing poses a threat to user privacy. Meters report fine-grained readings to the service provider, which potentially discloses sensitive information. For example, electricity smart meter readings reveal when users are at home and the electrical appliances they use [2], and electronic toll collection and "pay-as-you-drive" insurance reveal the driver's whereabouts [3,40,49].

In privacy-preserving billing protocols, meters do not send consumption measurements to the service provider. Instead, the computation of the bill is done locally and only the amount to be paid is revealed to the service provider.

Published online: 28 November 2016

Privacy-preserving billing protocols, in particular those which employ meters that are not tamper resistant, involve mechanisms to ensure that users report meter readings correctly, such as random spot checks in the electronic toll collection protocol in [3,37,43].

The protocols that use tamper-resistant meters either perform the bill calculation in the meter or outsource it to an untrusted platform to keep the meters simple. In [49], the bill calculation is performed inside the tamper-resistant meter. In contrast, in [47] the tamper-resistant meter outputs signed meter readings to a user application. At the end of a billing period, the user application employs the tariff policy sent by the service provider and the signed readings obtained from the meter to calculate the bill. The user application reveals to the service provider only the total bill, along with a proof that the computation of the bill is correct. This proof does not reveal any additional information on the meter readings. The approach in [47] has the advantage that it allows to minimize the trusted computing base and that it avoids the need to update tamper-resistant meters when the tariff policy changes. In addition, the mandatory deployment of smart meters in many countries implies that the purchase cost of a meter must be kept low. Therefore, it is advisable to keep meters as simple as possible. In [47] and in our protocol, meters are required to compute just digital signatures, but all the other computations are executed outside the meter. Practical implementations of these protocols have been shown in [17].

Our contribution We revisit the work of [47] and improve it in two ways. First, we generalize the security model in [47] to consider multiple meters and multiple users. Second, we propose a privacy-preserving billing protocol for our model that, in comparison with the protocol in [47], improves efficiency for policies described by splines.

The security model in [47] considered a setting where a meter communicates only with one user, and a user communicates only with one meter, i.e., there is a one-to-one relation between users and meters. This is insufficient for some smart metering applications. For example, consider a building where there is one meter per apartment that measures the water consumption in that apartment. Additionally, in the laundry room, there are one washing machine and one meter per apartment, and the meter measures the water consumption of the washing machine. In this example, the user needs to use meter readings from both meters to compute the water consumption bill.

As another example, consider a building with central heating. Each apartment is provided with a smart meter that measures the electricity consumption of its tenants. Additionally, another meter measures the electricity consumption of each of the tenants with respect to the central heating system. To model this setting adequately, it is necessary to both allow a meter to send meter readings to multiple users, and to allow a user to receive meter readings from more than one meter.

Of course, it is possible to use a protocol that considers only a one-to-one relation between users and meters in these examples. Simply, each meter–user pair is considered separately, and the user reports one separate bill for each meter. However, doing so does not achieve the same level of privacy offered in our model because the user reveals the price to be paid for the consumption at each meter, instead of revealing only the total price.

Therefore, we propose an ideal functionality $\mathcal{F}_{\text{BIL}}$ for privacy-preserving billing protocols that considers multiple meters and multiple users. In addition to that, $\mathcal{F}_{\text{BIL}}$ has the following main differences in comparison with the functionality for smart metering described in [47].

- $\mathcal{F}_{\text{BIL}}$ includes an interface through which the service provider sends a list of meters to a user at each billing period. The meter readings received from the meters in the list must be employed by the user to perform the bill calculation for that billing period.
- $\mathcal{F}_{\text{BIL}}$ includes an interface that allows meters to signal the end of a billing period and to report to the users the number of meter readings that were sent during the billing period. This necessary interface was omitted in the functionality in [47].
- $\mathcal{F}_{\text{BIL}}$ models explicitly the communication with the simulator $\mathcal{S}$. $\mathcal{S}$ needs this communication in order to provide a simulation for the adversary in the security proof.
- $\mathcal{F}_{\text{BIL}}$ allows any verifying party (and not just the service provider) to verify the bill to be paid. This may be useful in case of dispute between the meter and the service provider.
- $\mathcal{F}_{\text{BIL}}$ models the cases in which corrupt users collude with corrupt meters and/or with the service provider.

We propose a privacy-preserving billing protocol that realizes our functionality $\mathcal{F}_{\text{BIL}}$ and thus allows a meter to send meter readings to multiple users, and users to employ meter readings from multiple meters in the computation of a bill. In a nutshell, our protocol works as follows. At each billing period, the provider registers a signed tariff policy. Tariff policies are of the form $Y : (c, t) \rightarrow p$, where c is the consumption measurement, t is the time of consumption and p is the price. The provider also sends to each user a signed list of meters. Meters send signed meter readings to users and a signed "end of billing period" message that contains the number of meter readings sent from the meter to the user at that billing period. The user application calculates the bill and computes a zero-knowledge proof of knowledge of its correctness. This zero-knowledge proof involves proofs of signature possession that demonstrate that the correct tariff

policy is used to compute the price for each of the signed meter readings.

In [47], it is shown how to sign different types of tariff policies: a linear policy that multiplies each reading by a price per unit of consumption and a cumulative policy that divides the consumption range in intervals and applies a different price per unit to each interval. Additionally, it is mentioned that, in general, a tariff policy may be described by a polynomial for each interval. (Other functions can be approximated by polynomial splines.) Although the protocol in [47] provides efficient zero-knowledge proofs for the linear and cumulative policies, the cost of a zero-knowledge proof of a tariff policy described by a polynomial grows with the polynomial degree.

Our privacy-preserving billing protocol employs the same technique in [47] to sign linear and cumulative policies, and employs a new method for tariff policies described by splines. Consider the following tariff policy as example. A day is divided into L time intervals. For each time interval, the price to be paid for the consumption c is given by a spline:

$$Y(c,t) = \left\{ \begin{array}{ll} \Phi_1(c) & \text{if } t \in [t_1, t_2) \\ \vdots & \vdots \\ \Phi_L(c) & \text{if } t \in [t_L, t_{L+1}) \end{array} \right\}$$

Each spline $\Phi_l(c)$ ($l \in [1, L]$) is defined as follows.

$$\Phi_l(c) = \left\{ \begin{array}{ll} \phi_1(c) & \text{if } c \in [c_1, c_2) \\ \vdots & \vdots \\ \phi_M(c) & \text{if } c \in [c_M, c_{M+1}) \end{array} \right\}$$

Therefore, for a meter reading (c, t), the price to be paid is defined by the polynomial $\phi_m(c)$ such that $c \in [c_m, c_{m+1})$ that belongs to the spline $\Phi_l(c)$ associated with the time interval $[t_l, t_{l+1})$ such that $t \in [t_l, t_{l+1})$.

Alternatively, one can consider consumption bands, i.e., if a user's consumption is below a certain threshold, she may get a better price at peak hours. For each consumption band, the price to be paid at a certain time of day t is given by a spline where the polynomials take the time as input.

Our method to sign a tariff policy given by splines employs the polynomial commitment scheme of [29]. In a nutshell, the service provider computes polynomial commitments C to each of the polynomials in the tariff policy for the billing period bp. Additionally, the service provider computes, for each polynomial commitment, a signature on $[bp, C, t_{l-1}, t_l, c_{m-1}, c_m]$. The service provider sends the polynomial commitments and the signatures to the users, together with the polynomials. To prove in zero-knowledge that the price calculated for a meter reading is correct, the user evaluates the polynomial on input the consumption to compute the price and then proves possession of a witness for the polynomial commitment that shows that the price is the correct evaluation of the committed polynomial. The size of this proof of witness possession is independent of the polynomial degree. Additionally, the user proves possession of a signature on the polynomial commitment and proves that the values of consumption and time in the meter reading lie within the respective intervals in the signature.

Our use of polynomial commitments is somewhat different from their common use. In our scheme, the service provider computes polynomial commitments and sends them to the user together with the polynomials. Therefore, we do not need the hiding property of commitments. However, we need the binding property because the polynomial commitments are employed by the user to prove in zero-knowledge that prices are computed following the polynomials that define the tariff policy.

The reason why we use a polynomial commitment scheme is that it provides efficient selective opening, i.e., the commitment can be opened to an evaluation of the committed polynomial with cost independent of the polynomial degree. If a signature scheme is used, each of the coefficients of the polynomial needs to be signed as a separate message in the signature, and then, the cost of proving possession of the signature is linear in the polynomial degree.

We show that our protocol realizes $\mathcal{F}_{\text{BIL}}$. Unlike [47], we analyze all the possible collusion scenarios. Concretely, we analyze in detail the case in which the provider is corrupt, the case in which a subset of the users $\mathcal{U}$ are corrupt and the case in which the provider, a subset of the users and a subset of the meters $\mathcal{M}$ are corrupt. We analyze more briefly the case in which the provider $\mathcal{V}$ and a subset of the users are corrupt, the case in which a subset of the users $\mathcal{U}$ and a subset of the meters $\mathcal{M}$ are corrupt and the case in which the provider $\mathcal{V}$ and a subset of the meters $\mathcal{M}$ are corrupt.

For all the cases above, we consider Byzantine corruptions where a single adversary corrupts different parties and controls their behavior. Obviously, in this corruption model, when the provider and a meter are corrupt, there is no protocol that can prevent the provider from learning the meter readings input to the meter because both entities are controlled by the same adversary. For this reason, in Sect. 4.6.7, we consider a corruption model in which different adversaries, with no communication link between them, corrupt different parties. We show that, under such corruption model, our protocol prevents the corrupt meters from sending information about the meter readings to the provider. This is akin to showing that our protocol is collusion-free in the sense of [33].

Additionally, we consider the case in which the provider and the meters are corrupt, but do not have a side communication channel between them. We show that, in this case, our protocol is collusion-free in the sense of [33] and prevents corrupt meters from disclosing the meter readings to the provider or another corrupt verifying party.

We discuss how our protocol compares to other possible approaches for the design of privacy-preserving billing protocols in Sect. 5. Concretely, we discuss the use of regulations and codes of conduct, trusted parties, techniques to reduce variability, data anonymization methods, differential privacy, verifiable computing, and secure two-party and multi-party computation.

We note that our protocol is not only useful for billing, but, in general, allows to prove correctness of any computation on meter readings. This is important in settings such as smart metering, where meter readings are not only used for the sake of billing but also for consumption forecasting or profiling. For these other purposes, protocols that support complex computations on meter readings are necessary.

Outline of the paper In Sect. 2, we summarize the universally composable security framework and we describe our ideal functionality $\mathcal{F}_{\text{BIL}}$ for privacy-preserving billing. In Sect. 3, we describe the cryptographic building blocks that are employed by our protocol. We depict our protocol in Sect. 4. In Sect. 5, we discuss how our protocol compares to other possible approaches, and we conclude in Sect. 6.

2 Definition of privacy-preserving billing

In Sect. 2.1, we summarize the universal composability paradigm and describe the ideal functionalities for registration, common reference string and secure message transmission, which are employed in our protocols. In Sect. 2.2, we describe our ideal functionality for privacy-preserving billing.

2.1 Universal composability

The universal composability framework [9] is a framework for defining and analyzing the security of cryptographic protocols so that security is retained under arbitrary composition with other protocols. The security of a protocol is defined by means of an ideal protocol that carries out the desired task. In the ideal protocol, all the parties send their inputs to an ideal functionality $\mathcal{F}$ for the task. The ideal functionality computes locally the outputs of the parties and provides each party with its prescribed output.

The security of a protocol φ is analyzed by comparing the view of an environment $\mathcal{Z}$ in a real execution of φ against that of $\mathcal{Z}$ in the ideal protocol defined in $\mathcal{F}_\varphi$. The environment $\mathcal{Z}$ chooses the inputs of the parties and collects their outputs. In the real world, $\mathcal{Z}$ can communicate freely with an adversary $\mathcal{A}$ who controls the network as well as any corrupt parties. In the ideal world, $\mathcal{Z}$ interacts with dummy parties, who simply relay inputs and outputs between $\mathcal{Z}$ and $\mathcal{F}_\varphi$, and a simulator $\mathcal{S}$. We say that a protocol φ securely realizes $\mathcal{F}_\varphi$ if $\mathcal{Z}$ cannot distinguish the real world from the ideal world, i.e., $\mathcal{Z}$ cannot distinguish whether it is interacting with $\mathcal{A}$ and parties running protocol φ or with $\mathcal{S}$ and dummy parties relaying to $\mathcal{F}_\varphi$.

More formally, let $k \in \mathbb{N}$ denote the security parameter and $a \in \{0,1\}^*$ denote an input. Two binary distribution ensembles $X = \{X(k,a)\}_{k\in\mathbb{N},a\in\{0,1\}^*}$ and $Y = \{Y(k,a)\}_{k\in\mathbb{N},a\in\{0,1\}^*}$ are indistinguishable ($X \approx Y$) if for any $c, d \in \mathbb{N}$ there exists $k_0 \in \mathbb{N}$ such that for all $k > k_0$ and all $a \in \cup_{\kappa\leq k^d}\{0,1\}^\kappa$, $|\Pr[X(k,a)=1] - \Pr[Y(k,a)=1]| < k^{-c}$. Let $\text{REAL}_{\varphi,\mathcal{A},\mathcal{Z}}(k,a)$ denote the distribution given by the output of $\mathcal{Z}$ when executed on input a with $\mathcal{A}$ and parties running φ, and let $\text{IDEAL}_{\mathcal{F}_\varphi,\mathcal{S},\mathcal{Z}}(k,a)$ denote the output distribution of $\mathcal{Z}$ when executed on a with $\mathcal{S}$ and dummy parties relaying to $\mathcal{F}_\varphi$. We say that protocol φ securely realizes $\mathcal{F}_\varphi$ if, for all polynomial-time $\mathcal{A}$, there exists a polynomial-time $\mathcal{S}$ such that, for all polynomial-time $\mathcal{Z}$, $\text{REAL}_{\varphi,\mathcal{A},\mathcal{Z}} \approx \text{IDEAL}_{\mathcal{F}_\varphi,\mathcal{S},\mathcal{Z}}$.

A protocol $\varphi^{\mathcal{G}}$ securely realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model when φ is allowed to invoke the ideal functionality $\mathcal{G}$. Therefore, for any protocol ψ that securely realizes functionality $\mathcal{G}$, the composed protocol φ^ψ, which is obtained by replacing each invocation of an instance of $\mathcal{G}$ with an invocation of an instance of ψ, securely realizes $\mathcal{F}$.

When describing ideal functionalities, we use the following conventions:

Interface Naming Convention An ideal functionality can be invoked by using one or more interfaces. The name of a message in an interface consists of three fields separated by dots, e.g., reg.register.ini in the registration functionality described in Fig. 1. The first field indicates the name of the functionality and is the same in all the interfaces of the functionality. This first field is useful to distinguish between invocations of different functionalities in a hybrid protocol that employs two or more different functionalities. The second field indicates the kind of action performed by the functionality and is the same in all the messages that the functionality exchanges within the same interface. The third field distinguishes between the messages that belong to the same interface and can take four different values. A message $*.*.$ini is the incoming message received by the functionality, i.e., the message through which the interface is invoked. A message $*.*.$end is the outgoing message sent by the functionality, i.e., the message that ends the execution of the interface. The message $*.*.$sim is used by the functionality to send a message to the simulator, and the message $*.*.$rep is used to receive a message from the simulator.

Subsession identifiers Some interfaces in a functionality can be invoked more than once. When the functionality sends a message $*.*.$sim to the simulator in such an interface, a subsession identifier *ssid* is included in the

Functionality $\mathcal{F}_{\mathrm{REG}}^{\boxed{\mathsf{REG.Ver}}}$

1. On input $(\mathsf{reg.register.ini}, sid, v)$ from a party $\mathcal{T}$:
   - Abort if $sid \neq (\mathcal{T}, sid')$ or if there is a tuple $(sid, v', 0)$ stored.
   - $\boxed{\text{Abort if } 0 = \mathsf{REG.Ver}(v).}$
   - Store $(sid, v, 0)$.
   - Send $(\mathsf{reg.register.sim}, sid, v)$ to $\mathcal{S}$.

S. On input $(\mathsf{reg.register.rep}, sid)$ from the simulator $\mathcal{S}$:
   - Abort if $(sid, v, 0)$ is not stored or if $(sid, v, 1)$ is already stored.
   - Store $(sid, v, 1)$.
   - Send $(\mathsf{reg.register.end}, sid)$ to $\mathcal{T}$.

2. On input $(\mathsf{reg.retrieve.ini}, sid)$ from any party $\mathcal{P}$:
   - If $(sid, v, 1)$ is stored, set $v' \leftarrow v$, else set v' .
   - Create a fresh $ssid$ and store $(ssid, \mathcal{P}, v')$.
   - Send $(\mathsf{reg.retrieve.sim}, sid, ssid, v')$ to $\mathcal{S}$.

S. On input $(\mathsf{reg.retrieve.rep}, sid, ssid)$ from the simulator $\mathcal{S}$:
   - Abort if $(ssid, \mathcal{P}, v')$ is not stored.
   - Delete $ssid$ from $(ssid, \mathcal{P}, v')$.
   - Send $(\mathsf{reg.retrieve.end}, sid, v')$ to $\mathcal{P}$.

Functionality $\mathcal{F}_{\mathrm{SMT}}$

Parameterized by a leakage function $l : \{0,1\}^* \rightarrow \mathbb{N}$ that leaks the message length, $\mathcal{F}_{\mathrm{SMT}}$ works as follows:

1. On input $(\mathsf{smt.send.ini}, sid, m)$ from a party $\mathcal{T}$:
   - Abort if $sid \neq (\mathcal{T}, \mathcal{R}, sid')$.
   - Create a fresh $ssid$ and store $(ssid, \mathcal{R}, m)$.
   - Send $(\mathsf{smt.send.sim}, sid, ssid, l(m))$ to $\mathcal{S}$.

S. On input $(\mathsf{smt.send.rep}, sid, ssid)$ from $\mathcal{S}$:
   - Abort if $(ssid, \mathcal{R}, m)$ is not stored.
   - Delete $ssid$ from $(ssid, \mathcal{R}, m)$.
   - Send $(\mathsf{smt.send.end}, sid, m)$ to $\mathcal{R}$.

Functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$

Parameterized by a ppt algorithm $\mathsf{CRS.Setup}$, $\mathcal{F}_{\mathrm{CRS}}$ works as follows:

1. On input $(\mathsf{crs.get.ini}, sid)$ from any party $\mathcal{P}$:
   - If (sid, crs) is not stored, run $crs \leftarrow \mathsf{CRS.Setup}$ and store (sid, crs).
   - Create a fresh $ssid$ and store $(ssid, \mathcal{P})$.
   - Send $(\mathsf{crs.get.sim}, sid, ssid, crs)$ to $\mathcal{S}$.

S. On input $(\mathsf{crs.get.rep}, sid, ssid)$ from the simulator $\mathcal{S}$:
   - Abort if $(ssid, \mathcal{P})$ is not stored.
   - Delete $ssid$ from $(ssid, \mathcal{P})$.
   - Send $(\mathsf{crs.get.end}, sid, crs)$ to $\mathcal{P}$.

Fig. 1 Ideal functionalities $\mathcal{F}_{\mathrm{REG}}^{\boxed{\mathsf{REG.Ver}}}$, $\mathcal{F}_{\mathrm{SMT}}$ and $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$

message. The subsession identifier must also be included in the response $*.*.\mathsf{rep}$ sent by the simulator. The subsession identifier is used to identify the message $*.*.\mathsf{sim}$ to which the simulator replies with a message $*.*.\mathsf{rep}$. We note that, typically, the simulator in the security proof may not be able to provide an immediate answer to the functionality after receiving a message $*.*.\mathsf{sim}$. The reason is that the simulator typically needs to interact with the copy of the real adversary it runs in order to produce the message $*.*.\mathsf{rep}$, but the real adversary may not provide the desired answer, or may provide a delayed answer. In such cases, when the functionality sends more than one message $*.*.\mathsf{sim}$ to the simulator, the simulator may provide delayed replies, and the order of those replies may not follow the order of the received $*.*.\mathsf{sim}$ messages.

Aborts When we say that an ideal functionality $\mathcal{F}$ aborts after being activated with a message $(*, \ldots)$, we mean that $\mathcal{F}$ halts the execution of its program and sends a special abortion message $(*, \perp)$ to the party that invoked the functionality.

Network versus local communication The identity of an ITM instance (ITI) consists of a party identifier *pid* and a session identifier *sid*. A set of parties in an execution of a system of ITMs are a protocol instance if they have the same session identifier *sid*. ITIs can pass direct inputs to and outputs from "local" ITIs that have the same *pid*. An ideal functionality $\mathcal{F}$ has $pid = \perp$ and is considered local to all parties. An instance of $\mathcal{F}$ with the session identifier *sid* only accepts inputs from and passes outputs to machines with the same session identifier *sid*. Some functionalities require the session identifier to have some structure. Those functionalities check whether the session identifier possesses the required structure in the first message that invokes the functionality. For the next messages, the functionality implicitly checks that the session identifier equals the session identifier employed in the first message. Communication between ITIs with different party identifiers must take place over the network. The network is controlled by the adversary, meaning that he can arbitrarily delay, modify, drop or insert messages.

The conventions we use to describe of our ideal functionalities make them longer. The reason is that we have chosen not to omit any details, which are frequently omitted in the literature. There are two reasons why the descriptions of our functionalities are longer than usual.

- When our functionalities receive a message, we list all the reasons why the functionality must abort, including those related to the input message being malformed. Other functionalities in the literature omit these needed steps in their description.
- We describe in detail how the communication with the simulator takes place. In many ideal functionalities in the literature, after the functionality sends a message to the simulator, the functionality waits for the simulator to provide a response to that message. Similarly, many functionalities in the literature employ delayed outputs. However, in many cases, the simulator in the security

proof needs to interact with a copy of the real adversary in order to provide a response to the functionality. Therefore, the simulator may not be able to provide a response or may be able to do so only at a later stage. This means that many security proofs do not work because, when the functionality demands an immediate response from the simulator, the simulator is not able to provide it. To solve this problem, our functionalities do not require the simulator to provide an immediate response. Instead, our functionalities save their state, create a subsession identifier and call the simulator on input this subsession identifier. When the simulator sends a reply with a given subsession identifier, our functionalities recover the state and continue the computation. With this mechanism, our functionalities do not require the simulator to provide an immediate response.

It is possible to omit these operations in the description of a functionality and simply describe in a generic way that functionalities abort when an input message is malformed or that they save the state before calling the simulator and recover it when receiving a reply. However, our approach is less error-prone because it lists all the conditions for abortion and it shows what information needs to be saved and how this information is recovered.

Our protocol makes use of the functionality $\mathcal{F}_{\text{REG}}$ for key registration [9], $\mathcal{F}_{\text{SMT}}$ for secure message transmission [9] and $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$ [9] for common reference string generation. We describe these functionalities in Fig. 1. We also employ a variant $\mathcal{F}_{\text{REG}}^{\mathsf{REG.Ver}}$ of the registration functionality that is parameterized with a verification function $\mathsf{REG.Ver}$. We employ a box to indicate the steps that are only executed in this variant of $\mathcal{F}_{\text{REG}}$. We consider static corruptions only.

The functionalities $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{REG}}^{\mathsf{REG.Ver}}$ and $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$ are setup assumptions that we use in order to be able to provide a protocol that realizes our functionality $\mathcal{F}_{\text{BIL}}$ for privacy-preserving billing. In [9], it is explained that only very weakly security guarantees can be obtained in the bare model, i.e., without setup assumptions. In the real world, these setup assumptions can be realized by trusting certain parties, or alternatively by relying on certain physical phenomena. In the first case, to realize $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{REG}}^{\mathsf{REG.Ver}}$ and $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$, a protocol that follows the ideal world protocol defined by $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{REG}}^{\mathsf{REG.Ver}}$ and $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$ is employed, i.e., a trusted party in the real world executes the protocol.

In the case of $\mathcal{F}_{\text{SMT}}$, it is shown in [9] how this functionality can be realized by a protocol that uses a public key encryption scheme and an ideal functionality for authenticated communication. In [9], it is also shown that the ideal functionality for authenticated communication can be realized by a protocol that uses an existentially unforgeable signature scheme and the ideal functionality for registration $\mathcal{F}_{\text{REG}}$. Here, $\mathcal{F}_{\text{REG}}$ is a setup assumption that allows the realization of the ideal functionality for authenticated communication, which, as proven in [10], cannot be realized without setup assumptions.

2.2 Ideal functionality for privacy-preserving billing

We depict the ideal functionality $\mathcal{F}_{\text{BIL}}$ for privacy-preserving billing. $\mathcal{F}_{\text{BIL}}$ interacts with a provider $\mathcal{V}$, users $\mathcal{U}_i$, meters $\mathcal{M}_j$, and any verifying parties $\mathcal{P}$. The provider $\mathcal{V}$ creates billing periods bp. A billing period is not necessarily a period of time. More generally, it is an identifier that meters $\mathcal{M}_j$ associate to the meter readings that they output. The meter readings associated with the same billing period are used to compute the payment for that billing period.

The provider $\mathcal{V}$ associates to each billing period bp a tariff policy Y. The tariff policy $Y : (c, t) \rightarrow p$ is a function that takes in a consumption value c and the time of consumption t, and outputs the price to be paid p. $\mathcal{F}_{\text{BIL}}$ can easily be generalized to employ tariff policies that take as input more variables.

At a billing period bp, the provider $\mathcal{V}$ also associates each user $\mathcal{U}_i$ with a list of meters $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$. The meter readings output by those meters are employed by the user $\mathcal{U}_i$ to calculate the bill $p[bp]$ to be paid at the billing period bp.

A meter $\mathcal{M}_j$ can send meter readings to multiple users. A meter reading is a tuple $(\mathcal{U}_i, bp, c, t)$. At the end of a billing period bp, $\mathcal{M}_j$ also sends a user $\mathcal{U}_i$ the number of meter readings $N[\mathcal{M}_j, bp]$ that $\mathcal{M}_j$ sent to $\mathcal{U}_i$ during the billing period bp.

A user $\mathcal{U}_i$ obtains the tariff policy Y and the list of meters $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$ for the billing period bp. The user $\mathcal{U}_i$ also gets meter readings from multiple meters. In order to compute the bill $p[bp]$, $\mathcal{U}_i$ employs the meter readings received from the meters in the list $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$. $\mathcal{U}_i$ applies the tariff policy Y to each meter reading in order to compute a price p. The prices for all the meter readings are added in order to obtain the bill $p[bp]$.

Any verifying party $\mathcal{P}$ receives the bill $p[bp]$ from a user. $\mathcal{P}$ could be the provider $\mathcal{V}$ but, in general, is any party that verifies the correctness of $p[bp]$.

The interaction between the functionality $\mathcal{F}_{\text{BIL}}$ and the provider $\mathcal{V}$, the users $\mathcal{U}_i$, the meters $\mathcal{M}_j$ and the verifying parties $\mathcal{P}$ takes place through the following interfaces:

1. The provider $\mathcal{V}$ uses the $\mathsf{bil.policy}.*$ interface to send the pricing policy Y associated with the billing period bp.
2. The provider $\mathcal{V}$ uses the $\mathsf{bil.listmeters}.*$ interface to send the list of meters $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$ associated with a user $\mathcal{U}_i$ at the billing period bp.
3. A meter $\mathcal{M}_j$ uses the $\mathsf{bil.consumption}.*$ interface to send a meter reading (c, t) for the billing period bp to a user $\mathcal{U}_i$.

4. A meter $\mathcal{M}_j$ uses the bil.period.$*$ interface to send to a user $\mathcal{U}_i$ the number of meter readings $N[\mathcal{M}_j, bp]$ that $\mathcal{M}_j$ sent to $\mathcal{U}_i$ during the billing period bp.
5. A user $\mathcal{U}_i$ employs the bil.payment.$*$ interface to send to any verifying party $\mathcal{P}$ the bill $p[bp]$ for the billing period bp. $\mathcal{U}_i$ also discloses to the provider $\mathcal{V}$ the number of meter readings $N[\mathcal{M}_j, bp]$ obtained from each of the meters $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$.

$\mathcal{F}_{\text{BIL}}$ employs a table T to store meter readings. T stores entries of the form $(\mathcal{M}_j, \mathcal{U}_i, bp, c, t, b)$. $\mathcal{M}_j$ is the identifier of the meter that outputs the meter reading. $\mathcal{U}_i$ is the identifier of the user that receives the meter reading. bp denotes the billing period, c is the consumption value, and t is the time of consumption. The bit $b = 0$ indicates that the reading was not received by the user, while $b = 1$ indicates that the reading was received by the user.

$\mathcal{F}_{\text{BIL}}$ has the following main differences in comparison with the functionality for smart metering described in [47].

- $\mathcal{F}_{\text{BIL}}$ interacts with multiple users and multiple meters, while the functionality in [47] only considers one meter and one user. Furthermore, $\mathcal{F}_{\text{BIL}}$ allows a meter to send meter readings to multiple users, and users receive meter readings from multiple meters. Therefore, $\mathcal{F}_{\text{BIL}}$ is applicable to a wider variety of billing settings.
- $\mathcal{F}_{\text{BIL}}$ includes an interface through which the service provider sends a list of meters to a user at each billing period. The meter readings received from the meters in the list must be employed by the user to perform the bill calculation for that billing period.
- $\mathcal{F}_{\text{BIL}}$ includes an interface bil.period.$*$, which allow meters to signal the end of a billing period and to report to the users the number of meter readings that were sent during the billing period. This necessary interface was omitted in the functionality in [47].
- $\mathcal{F}_{\text{BIL}}$ models explicitly the communication with the simulator $\mathcal{S}$. $\mathcal{S}$ needs this communication in order to provide a simulation for the adversary in the security proof.
- $\mathcal{F}_{\text{BIL}}$ allows any verifying party to receive the bill to be paid. This may be useful in case of dispute between the meter and the service provider.
- $\mathcal{F}_{\text{BIL}}$ models the cases in which corrupt users collude with corrupt meters and/or with the service provider. In the functionality in [47], a corrupt meter was not considered because they were assumed to be tamper resistant. A collusion of a corrupt provider with a corrupt user was also not considered because it lacked practical interest. However, when considering multiple meters and users as $\mathcal{F}_{\text{BIL}}$ does, $\mathcal{F}_{\text{BIL}}$ must still provide some security guarantees for honest users in the case in which some meters, some users and the provider are corrupt. For example, $\mathcal{F}_{\text{BIL}}$ guarantees that such a collusion is prevented from reporting a bill calculation to any verifying party $\mathcal{P}$ on behalf of an honest user. We note that, when a user colludes with the provider or with a meter included in the list of meters for a billing period, the price to be paid is not computed by $\mathcal{F}_{\text{BIL}}$, but is input by the simulator $\mathcal{S}$ and thus may not be correct.

We now discuss the five interfaces of the ideal functionality $\mathcal{F}_{\text{BIL}}$, which we depict in Figs. 2 and 3.

1. The provider $\mathcal{V}$ invokes the bil.policy.ini message on input a billing period bp and a tariff policy Y. The restriction that the provider's identity must be included in the session identifier $sid = (\mathcal{V}, sid')$ guarantees that each provider can initialize its own instance of the functionality. This check is implicitly done in the other interfaces. The functionality also checks that the billing period and the tariff policy belong to their respective universes of allowed inputs. $\mathcal{F}_{\text{BIL}}$ performs similar checks on the data received as input through the other interfaces. $\mathcal{F}_{\text{BIL}}$ also aborts if a policy for that billing period was already received through the bil.policy.ini message. Otherwise $\mathcal{F}_{\text{BIL}}$ stores bp and Y and sends bp and Y to the simulator $\mathcal{S}$ through the bil.policy.sim message.
   After being triggered by the simulator $\mathcal{S}$ through the bil.policy.rep message on input a billing period bp, $\mathcal{F}_{\text{BIL}}$ aborts if the policy for that billing period was not received through the bil.policy.ini message or if the registration of the policy was already finalized. To realize this feature in any construction for $\mathcal{F}_{\text{BIL}}$, the registration functionality in Sect. 2.1 can be employed. If $\mathcal{F}_{\text{BIL}}$ does not abort, $\mathcal{F}_{\text{BIL}}$ stores the policy Y for the billing period bp.
2. The provider $\mathcal{V}$ invokes the bil.listmeters.ini message on input a billing period bp, a user identifier $\mathcal{U}_i$ and a list of meter identifiers $(\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m})$. $\mathcal{F}_{\text{BIL}}$ aborts if a list of meters for the same user and billing period was already received as input before. We note that if the provider $\mathcal{V}$ and the user $\mathcal{U}_i$ are corrupt, $\mathcal{S}$ can change the list of meters used for a payment through the message bil.payment.rep. Otherwise, $\mathcal{F}_{\text{BIL}}$ records that a list of meters for that user at that billing period has been sent. $\mathcal{F}_{\text{BIL}}$ creates a subsession identifier and sends $\mathcal{U}_i$ to the simulator $\mathcal{S}$ through the message bil.listmeters.sim. Since bp and $(\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m})$ are not revealed to $\mathcal{S}$, any construction that realizes $\mathcal{F}_{\text{BIL}}$ would need to employ a communication channel that prevents bp and $(\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m})$ from being disclosed to $\mathcal{S}$. The functionality $\mathcal{F}_{\text{SMT}}$ in Sect. 2.1 fulfills this property.
   After being triggered by $\mathcal{S}$ through the bil.listmeters.rep message, $\mathcal{F}_{\text{BIL}}$ aborts if the subsession identifier does not exist. If $\mathcal{F}_{\text{BIL}}$ does not abort, $\mathcal{F}_{\text{BIL}}$ stores the meter list and sends the meter list to the user $\mathcal{U}_i$.

Functionality $\mathcal{F}_{\text{BIL}}$: Interfaces bil.policy.$*$**,** bil.listmeters.$*$ **and** bil.consumption.$*$

$\mathcal{F}_{\text{BIL}}$ is parameterized by a universe of policies U_y, a universe of consumptions U_c, a universe of times U_t, a universe of billing periods U_{bp}, and a maximum size M_{max} for the meter lists. $\mathcal{F}_{\text{BIL}}$ interacts with a provider $\mathcal{V}$, users $\mathcal{U}_i$, meters $\mathcal{M}_j$ and verifying parties $\mathcal{P}$.

1. On input (bil.policy.ini, sid, bp, Y) from $\mathcal{V}$:
   - Abort if $sid \neq (\mathcal{V}, sid')$, or if $bp \notin U_{bp}$, or if $Y \notin U_y$, or if $(sid, bp', Y', 0)$ such that $bp' = bp$ is already stored.
   - Store $(sid, bp, Y, 0)$.
   - Send (bil.policy.sim, sid, bp, Y) to $\mathcal{S}$.

S. On input (bil.policy.rep, sid, bp) from $\mathcal{S}$:
   - Abort if $(sid, bp, Y, 0)$ is not stored or if $(sid, bp, Y, 1)$ is already stored.
   - Store $(sid, bp, Y, 1)$.
   - Parse sid as $(\mathcal{V}, sid')$.
   - Send (bil.policy.end, sid) to $\mathcal{V}$.

2. On input (bil.listmeters.ini, sid, bp,$\mathcal{U}_i$, $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$) from $\mathcal{V}$:
   - Abort if $bp \notin U_{bp}$, or if $\mathcal{U}_i$ is not a user identifier, or if $m > M_{max}$, or if, for $k = 1$ to m, $\mathcal{M}_{j_m}$ is not a meter identifier.
   - Abort if $(sid, bp', \mathcal{U}_i', \mathcal{M}_{j_1}', \ldots, \mathcal{M}_{j_m}', 0)$ such that $bp = bp'$ and $\mathcal{U}_i = \mathcal{U}_i'$ is already stored.
   - Store $(sid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, 0)$.
   - Create a fresh $ssid$ and store $(ssid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m})$.
   - Send (bil.listmeters.sim, sid, $ssid$, $\mathcal{U}_i$) to $\mathcal{S}$.

S. On input (bil.listmeters.rep, sid, $ssid$) from $\mathcal{S}$:
   - Abort if $(ssid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m})$ is not stored.
   - Store $(sid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, 1)$.
   - Delete $(ssid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m})$.
   - Send (bil.listmeters.end, sid, bp, $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$) to the user $\mathcal{U}_i$.

3. On input (bil.consumption.ini, sid,$\mathcal{U}_i$, bp, c, t) from the meter $\mathcal{M}_j$:
   - Abort if $\mathcal{U}_i$ is not a user identifier, or if $bp \notin U_{bp}$, or if $c \notin U_c$, or if $t \notin U_t$.
   - Abort if $(sid, \mathcal{M}_j', \mathcal{U}_i', bp', N[\mathcal{M}_j, bp], 0)$ such that $\mathcal{M}_j' = \mathcal{M}_j, \mathcal{U}_i' = \mathcal{U}_i$ and $bp' = bp$ is already stored.
   - Store $(\mathcal{M}_j, \mathcal{U}_i, bp, c, t, 0)$ in Table T.
   - Create a fresh $ssid$ and store $(ssid, \mathcal{M}_j, \mathcal{U}_i, bp, c, t)$.
   - Send (bil.consumption.sim, sid, $ssid$, $\mathcal{M}_j$, $\mathcal{U}_i$) to $\mathcal{S}$.

S. On input (bil.consumption.rep, sid, $ssid$) from $\mathcal{S}$:
   - Abort if $(ssid, \mathcal{M}_j, \mathcal{U}_i, bp, c, t)$ is not stored.
   - Delete $ssid$ from $(\mathcal{M}_j, \mathcal{U}_i, bp, c, t)$.
   - Abort if $(sid, \mathcal{M}_j', \mathcal{U}_i', bp', N[\mathcal{M}_j, bp], 1)$ such that $\mathcal{M}_j' = \mathcal{M}_j, \mathcal{U}_i' = \mathcal{U}_i$ and $bp' = bp$ is already stored.
   - Replace $(\mathcal{M}_j, \mathcal{U}_i, bp, c, t, 0)$ by $(\mathcal{M}_j, \mathcal{U}_i, bp, c, t, 1)$ in Table T.
   - Send (bil.consumption.end, sid, $\mathcal{M}_j$, bp, c, t) to $\mathcal{U}_i$.

Fig. 2 $\mathcal{F}_{\text{BIL}}$: interfaces bil.policy.$*$, bil.listmeters.$*$ and bil.consumption.$*$

3. A meter $\mathcal{M}_j$ invokes the bil.consumption.ini message on input a user identifier $\mathcal{U}_i$, a billing period bp, a consumption value c and a time t. $\mathcal{F}_{\text{BIL}}$ aborts if the meter $\mathcal{M}_j$ had already sent an end of period message for the billing period bp through the bil.period.ini message. We note that if the meter $\mathcal{M}_j$ and the user $\mathcal{U}_i$ are corrupt, the simulator can input an incorrect bill $p[bp]$ through the bil.payment.rep message. Otherwise, $\mathcal{F}_{\text{BIL}}$ stores the meter reading sent by the meter in the table T. $\mathcal{F}_{\text{BIL}}$ creates a subsession identifier and sends the meter identifier $\mathcal{M}_j$ and the user identifier $\mathcal{U}_i$ to the simulator $\mathcal{S}$. The values c and t are not disclosed. Therefore, any construction that realizes $\mathcal{F}_{\text{BIL}}$ would need to employ a secure channel such as $\mathcal{F}_{\text{SMT}}$.
   After being triggered by the simulator $\mathcal{S}$ through the message bil.consumption.rep, $\mathcal{F}_{\text{BIL}}$ aborts if the subsession identifier is not stored. $\mathcal{F}_{\text{BIL}}$ also aborts if the end of billing period message has already been sent to the user through a bil.period.end message. We note that it is possible that $\mathcal{F}_{\text{BIL}}$ receives a meter reading through a bil.consumption.ini message before the end of billing period message is received through a bil.period.ini message, but the bil.consumption.rep message for that meter reading is received after the end of billing period message is sent to the user. If $\mathcal{F}_{\text{BIL}}$ does not abort, $\mathcal{F}_{\text{BIL}}$ indicates in the table T that the meter reading is received by the user and sends $\mathcal{M}_j$, bp, c and t to the user $\mathcal{U}_i$.
4. A meter $\mathcal{M}_j$ invokes the bil.period.ini message on input a user identifier $\mathcal{U}_i$ and a billing period bp. $\mathcal{F}_{\text{BIL}}$ checks the validity of the input. $\mathcal{F}_{\text{BIL}}$ aborts if the message bil.period.ini was already sent for the same user, meter

Functionality $\mathcal{F}_{\mathrm{BIL}}$: Interfaces bil.period.$*$ **and** bil.payment.$*$

4. On input (bil.period.ini, $sid, \mathcal{U}_i, bp$) from $\mathcal{M}_j$:
   - Abort if $\mathcal{U}_i$ is not a user identifier, or if $bp \notin U_{bp}$.
   - Abort if $(sid, \mathcal{M}'_j, \mathcal{U}'_i, bp', N[\mathcal{M}_j, bp], 0)$ such that $\mathcal{M}'_j = \mathcal{M}_j, \mathcal{U}'_i = \mathcal{U}_i$ and $bp' = bp$ is already stored.
   - Set $N[\mathcal{M}_j, bp]$ to the number of entries $(\mathcal{M}'_j, \mathcal{U}'_i, bp', c', t', b)$ in Table T such that $\mathcal{M}'_j = \mathcal{M}_j, \mathcal{U}'_i = \mathcal{U}_i$ and $bp' = bp$.
   - Store $(sid, \mathcal{M}_j, \mathcal{U}_i, bp, N[\mathcal{M}_j, bp], 0)$.
   - Create a fresh $ssid$ and store $(ssid, \mathcal{M}_j, \mathcal{U}_i, bp, N[\mathcal{M}_j, bp])$.
   - Send (bil.period.sim, $sid, ssid, \mathcal{M}_j, \mathcal{U}_i$) to $\mathcal{S}$.

S. On input (bil.period.rep, $sid, ssid$) from $\mathcal{S}$:
   - Abort if $(ssid, \mathcal{M}_j, \mathcal{U}_i, bp, N[\mathcal{M}_j, bp])$ is not stored.
   - Delete $ssid$ from $(ssid, \mathcal{M}_j, \mathcal{U}_i, bp, N[\mathcal{M}_j, bp])$.
   - Set $N'[\mathcal{M}_j, bp]$ to the number of entries $(\mathcal{M}'_j, \mathcal{U}'_i, bp', c', t', b)$ in Table T such that $\mathcal{M}'_j = \mathcal{M}_j, \mathcal{U}'_i = \mathcal{U}_i$, $bp' = bp$, and $b = 1$.
   - Abort if $N'[\mathcal{M}_j, bp] \neq N[\mathcal{M}_j, bp]$.
   - Store $(sid, \mathcal{M}_j, \mathcal{U}_i, bp, N[\mathcal{M}_j, bp], 1)$.
   - Send (bil.period.end, $sid, bp, \mathcal{M}_j, N[\mathcal{M}_j, bp]$) to $\mathcal{U}_i$.

5. On input (bil.payment.ini, $sid, \mathcal{P}, bp$) from $\mathcal{U}_i$:
   - Abort if $\mathcal{P}$ is not a valid party identifier, of if $bp \notin U_{bp}$.
   - Abort if $(sid, bp', Y', 1)$ such that $bp' = bp$ is not stored.
   - If $\mathcal{U}_i$ is honest or if $\mathcal{V}$ is honest, abort if $(sid, bp', \mathcal{U}'_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, 1)$ such that $bp' = bp$ and $\mathcal{U}'_i = \mathcal{U}_i$ is not stored.
   - For $k = 1$ to m, if $\mathcal{U}_i$ is honest or if $\mathcal{V}$ and $\mathcal{M}_{j_k}$ are honest, abort if a tuple $(sid, \mathcal{M}'_j, \mathcal{U}'_i, bp', N[\mathcal{M}_j, bp], 1)$ such that $\mathcal{M}'_j = \mathcal{M}_{j_k}$, $bp' = bp$ and $\mathcal{U}'_i = \mathcal{U}_i$ is not stored.
   - Create a fresh $ssid$ and store $(ssid, \mathcal{U}_i, \mathcal{P}, bp)$.
   - Send (bil.payment.sim, $sid, ssid, \mathcal{U}_i, \mathcal{P}$) to $\mathcal{S}$.

S. On input (bil.payment.rep, $sid, ssid$), if either $\mathcal{U}_i$ is honest or if $\mathcal{V}$ and the meters in the list $(sid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, 1)$ are honest, or else on input (bil.payment.rep, $sid, ssid, p[bp], \langle \mathcal{M}_{j_k}, N[\mathcal{M}_{j_k}, bp] \rangle_{k=1}^{m}$) from $\mathcal{S}$:
   - Abort if $(ssid, \mathcal{U}_i, \mathcal{P}, bp)$ is not stored.
   - Delete $ssid$ from $(ssid, \mathcal{U}_i, \mathcal{P}, bp)$.
   - Abort if $\mathcal{U}_i$ and $\mathcal{V}$ are corrupt and $m > M_{max}$.
   - If $\mathcal{U}_i$ or $\mathcal{V}$ are honest, retrieve $(\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m})$ from the tuple $(sid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, 1)$, else employ the list of meters sent by $\mathcal{S}$.
   - For $k = 1$ to m, if $\mathcal{M}_{j_k}$ is honest, abort if a tuple $(sid, \mathcal{M}'_j, \mathcal{U}'_i, bp', N[\mathcal{M}'_j, bp], 1)$ such that $\mathcal{M}'_j = \mathcal{M}_{j_k}$, $bp' = bp$ and $\mathcal{U}'_i = \mathcal{U}_i$ is not stored.
   - If either $\mathcal{U}_i$ is honest or if $\mathcal{V}$ and the meters in the retrieved list are honest, for $k = 1$ to m, do the following:
     - Set $p[\mathcal{M}_{j_k}, bp] = 0$.
     - Retrieve $N[\mathcal{M}'_j, bp]$ from the tuple $(sid, \mathcal{M}'_j, \mathcal{U}'_i, bp', N[\mathcal{M}'_j, bp], 1)$ such that $\mathcal{M}'_j = \mathcal{M}_{j_k}$, $bp' = bp$ and $\mathcal{U}'_i = \mathcal{U}_i$.
     - Retrieve all the $N[\mathcal{M}'_j, bp]$ tuples $(\mathcal{M}'_j, \mathcal{U}'_i, bp', c, t, 1)$ in Table T such that $\mathcal{M}'_j = \mathcal{M}_{j_k}, \mathcal{U}'_i = \mathcal{U}_i$ and $bp' = bp$.
     - For $n = 1$ to $N[\mathcal{M}_{j_k}, bp]$, set $p[\mathcal{M}_{j_k}, bp] = p[\mathcal{M}_{j_k}, bp] + Y(c[k, n], t[k, n])$.
   - If either $\mathcal{U}_i$ is honest or if $\mathcal{V}$ and the meters in the retrieved list are honest, set $p[bp] = p[\mathcal{M}_{j_1}, bp] + \ldots + p[\mathcal{M}_{j_m}, bp]$, else employ the value $p[bp]$ sent by $\mathcal{S}$.
   - For $k = 1$ to m, if $\mathcal{U}_i$ or $\mathcal{M}_{j_k}$ (in the retrieved meter list) are honest, retrieve $N[\mathcal{M}'_j, bp]$ from the tuple $(sid, \mathcal{M}'_j, \mathcal{U}'_i, bp', N[\mathcal{M}'_j, bp], 1)$ such that $\mathcal{M}'_j = \mathcal{M}_{j_k}$, $bp' = bp$ and $\mathcal{U}'_i = \mathcal{U}_i$, else employ the value $N[\mathcal{M}_{j_k}, bp]$ sent by $\mathcal{S}$ or abort if this value is lower than 0.
   - Send (bil.payment.end, $sid, \mathcal{U}_i, bp, p[bp], \mathcal{M}_{j_1}, N[\mathcal{M}_{j_1}, bp], \ldots, \mathcal{M}_{j_m}, N[\mathcal{M}_{j_m}, bp]$) to $\mathcal{P}$.

Fig. 3 $\mathcal{F}_{\mathrm{BIL}}$: interfaces bil.period.$*$ and bil.payment.$*$

and billing period. Else $\mathcal{F}_{\mathrm{BIL}}$ calculates the number $N[\mathcal{M}_j, bp]$ of meter readings that $\mathcal{M}_j$ sent to $\mathcal{U}_i$ at the billing period bp. We note that, if the meter $\mathcal{M}_j$ and the user $\mathcal{U}_i$ are corrupt, the simulator can change the number of meter readings for that billing period through the bil.payment.rep message. $\mathcal{F}_{\mathrm{BIL}}$ creates a subsession identifier and sends the meter identifier $\mathcal{M}_j$ and the user identifier $\mathcal{U}_i$ to the simulator $\mathcal{S}$.

After being triggered by $\mathcal{S}$ through the bil.period.rep message, $\mathcal{F}_{\mathrm{BIL}}$ aborts if the subsession identifier is not stored. $\mathcal{F}_{\mathrm{BIL}}$ calculates the number of meter readings received by $\mathcal{U}_i$ from $\mathcal{M}_j$ at that billing period and aborts if that number does not equal the number of meter readings sent by $\mathcal{M}_j$ to $\mathcal{U}_i$. If $\mathcal{F}_{\mathrm{BIL}}$ does not abort, $\mathcal{F}_{\mathrm{BIL}}$ stores $N[\mathcal{M}_j, bp]$ and sends $N[\mathcal{M}_j, bp]$ to $\mathcal{U}_i$.

5. A user $\mathcal{U}_i$ invokes the bil.payment.ini message on input the identifier of a verifying party $\mathcal{P}$ and a billing period bp. $\mathcal{F}_{\mathrm{BIL}}$ aborts if the tariff policy is not stored. $\mathcal{F}_{\mathrm{BIL}}$ also aborts if $\mathcal{U}_i$ or $\mathcal{V}$ are honest, but the list of meters for the billing period bp is not stored. $\mathcal{F}_{\mathrm{BIL}}$ does not

abort for this reason when $\mathcal{U}_i$ and $\mathcal{V}$ are corrupt because, in that case, $\mathcal{S}$ is allowed to input another list through the bil.payment.rep message. $\mathcal{F}_{\text{BIL}}$ also aborts if $\mathcal{U}_i$ is honest or if $\mathcal{V}$ and any of the meters in the list are honest, but the end of period message from that meter was not received by the user $\mathcal{U}_i$. We note that $\mathcal{F}_{\text{BIL}}$ does not abort for that reason when the meter is honest, but the user and the provider are corrupt. The reason is that, when the user and the provider are corrupt, $\mathcal{S}$ may send a different list of meters through the bil.payment.rep message. If $\mathcal{F}_{\text{BIL}}$ does not abort, $\mathcal{F}_{\text{BIL}}$ creates a subsession identifier and sends the user identifier $\mathcal{U}_i$ and the party identifier $\mathcal{P}$ to the simulator $\mathcal{S}$.

When the simulator $\mathcal{S}$ invokes the bil.payment.rep message, we distinguish two cases.

User honest or provider and meters honest $\mathcal{S}$ sends no input to $\mathcal{F}_{\text{BIL}}$ through the bil.payment.rep message. $\mathcal{F}_{\text{BIL}}$ aborts if the subsession identifier is not stored. Otherwise, $\mathcal{F}_{\text{BIL}}$ computes the bill $p[bp]$ as follows. For each of the meters in the list of meters that the provider sent to the user for the billing period bp, $\mathcal{F}_{\text{BIL}}$ takes the meter readings that the meter sent to the user at that billing period. $\mathcal{F}_{\text{BIL}}$ applies the policy for the billing period bp to each of the meter readings to obtain a price. The prices associated with a meter are summed up to get a price $p[\mathcal{M}_{j_k}, bp]$ for the meter readings sent by the meter $\mathcal{M}_{j_k}$. Finally, the prices corresponding to each meter are summed up to get the bill $p[bp]$. $\mathcal{F}_{\text{BIL}}$ sends to the party $\mathcal{P}$ the bill, the billing period and the user identifier along with the meter list and the number of meter readings from each meter.
User corrupt and provider or meters corrupt $\mathcal{S}$ sends to $\mathcal{F}_{\text{BIL}}$ a price, a list of meters and a counter of meter readings for each meter. If the provider is honest, $\mathcal{F}_{\text{BIL}}$ disregards the list of meters sent by $\mathcal{S}$ and uses instead the list that the functionality stores for that billing period. For each of the meters in the list, if the user or the meter are honest, $\mathcal{F}_{\text{BIL}}$ disregards the counter of meter readings sent by $\mathcal{S}$ and uses instead the one the functionality stores. $\mathcal{F}_{\text{BIL}}$ outputs the price, the billing period and the user identifier along with the list of meters and counter of meter readings from each meter.

We note that disclosing to the verifying party $\mathcal{P}$ the number of meter readings from each meter along with the bill may reveal sensitive information about the user. It is easy to modify $\mathcal{F}_{\text{BIL}}$ so that this information is not disclosed. However, the constructions that realize such a functionality would be less efficient.

3 Technical preliminaries

3.1 Non-interactive zero-knowledge proofs of knowledge

Let R be a polynomial-time computable binary relation. For tuples $(wit, ins) \in R$ we call wit the witness and ins the instance. Let L be the NP language consisting of the instances ins for which there exist witnesses wit such that $(wit, ins) \in R$. A non-interactive zero-knowledge proof of knowledge (NIPK) system for the relation R consists of three algorithms PKSetup, PKProve and PKVerify. On input a security parameter 1^k, $\mathsf{PKSetup}(1^k)$ outputs the parameters par_{pk}. The algorithm $\mathsf{PKProve}(par_{pk}, wit, ins)$ checks whether $(wit, ins) \in R$ and in that case outputs a proof π. $\mathsf{PKVerify}(par_{pk}, ins, \pi)$ outputs 1 if π is a valid proof that $ins \in L$ or 0 if that is not the case.

Definition 1 A NIPK system must fulfill the following completeness, extractability and zero-knowledge properties.

Completeness Completeness requires that the verification algorithm PKVerify accepts the proofs computed by the algorithm PKProve. More formally, for all $(wit, ins) \in R$, the completeness property is defined as follows.

$$\Pr\left[\begin{array}{l} par_{pk} \xleftarrow{\$} \mathsf{PKSetup}(1^k); \\ \pi \xleftarrow{\$} \mathsf{PKProve}(par_{pk}, wit, ins) : \\ 1 = \mathsf{PKVerify}(par_{pk}, ins, \pi) \end{array}\right] = 1$$

Extractability The extractability property requires the existence of a knowledge extractor $(\mathcal{E}_1, \mathcal{E}_2)$. $\mathcal{E}_1(1^k)$ outputs parameters par_{pk} and a trapdoor td_e such that par_{pk} is indistinguishable from the output of $\mathsf{PKSetup}(1^k)$. More formally, for all polynomial-time adversaries $\mathcal{A}$:

$$\Pr[par_{pk} \xleftarrow{\$} \mathsf{PKSetup}(1^k) : 1 = \mathcal{A}(par_{pk})] \approx$$
$$\Pr[(par_{pk}, td_e) \xleftarrow{\$} \mathcal{E}_1(1^k) : 1 = \mathcal{A}(par_{pk})]$$

For all polynomial-time adversaries $\mathcal{A}, \mathcal{E}_2$ extracts wit from a valid proof with overwhelming probability. More formally,

$$\Pr\left[\begin{array}{l} (par_{pk}, td_e) \xleftarrow{\$} \mathcal{E}_1(1^k); \\ (ins, \pi) \xleftarrow{\$} \mathcal{A}(par_{pk}, td_e); \\ wit \leftarrow \mathcal{E}_2(par_{pk}, td_e, ins, \pi) : \\ 1 = \mathsf{PKVerify}(par_{pk}, ins, \pi) \wedge \\ (ins, wit) \notin R \end{array}\right] \leq \epsilon(k)$$

Zero-knowledge Zero-knowledge requires that there exists a simulator $(\mathcal{S}_1, \mathcal{S}_2)$ such that, for all polynomial-time adversaries $\mathcal{A}$:

$$\Pr[par_{pk} \xleftarrow{\$} \mathsf{PKSetup}(1^k) : 1 = \mathcal{A}(par_{pk})^{\leftrightarrow \mathcal{O}_p(par_{pk},\cdot,\cdot)}] \approx$$
$$\Pr[(par_{pk}, td_s) \xleftarrow{\$} \mathcal{S}_1(1^k) : 1 = \mathcal{A}(par_{pk})^{\leftrightarrow \mathcal{S}(par_{pk},td_s,\cdot,\cdot)}]$$

The oracle $\mathcal{O}_p(par_{pk}, wit, ins)$ executes the algorithm $\mathsf{PKProve}(par_{pk}, wit, ins)$ and returns its output. (We recall that $\mathsf{PKProve}$ only outputs a proof if $(wit, ins) \in R$.) $\mathcal{S}(par_{pk}, td_s, wit, ins)$ runs $\mathcal{S}_2(par_{pk}, td_s, ins)$ and returns its output if $(wit, ins) \in R$, else returns failure.

3.2 Signature schemes

A signature scheme consists of the algorithms KeyGen, Sign and VfSig. Algorithm $\mathsf{KeyGen}(1^k)$ outputs a secret key sk and a public key pk, which include a description of the message space $\mathcal{M}$. $\mathsf{Sign}(sk, m)$ outputs a signature s on a message $m \in \mathcal{M}$. $\mathsf{VfSig}(pk, s, m)$ outputs 1 if s is a valid signature on m and 0 otherwise. This definition can be extended to blocks of messages $\bar{m} = (m_1, \ldots, m_n)$. In this case, $\mathsf{KeyGen}(1^k, n)$ receives the maximum number of messages as input.

Definition 2 A signature scheme must fulfill the following correctness and existential unforgeability properties [21].

Correctness Correctness ensures that the algorithm VfSig accepts the signatures created by the algorithm Sign on input a secret key computed by the algorithm KeyGen. More formally, correctness is defined as follows.

$$\Pr\left[\begin{array}{l}(sk, pk) \xleftarrow{\$} \mathsf{KeyGen}(1^k);\ m \xleftarrow{\$} \mathcal{M};\\ s \xleftarrow{\$} \mathsf{Sign}(sk, m) :\ 1 = \mathsf{VfSig}(pk, s, m)\end{array}\right] = 1$$

Existential Unforgeability The property of existential unforgeability ensures that it is not feasible to output a signature on a message without knowledge of the secret key or of another signature on that message. Let $\mathcal{O}_s$ be an oracle that, on input sk and a message $m \in \mathcal{M}$, outputs $\mathsf{Sign}(sk, m)$, and let S_s be a set that contains the messages sent to $\mathcal{O}_s$. More formally, for any ppt adversary $\mathcal{A}$, existential unforgeability is defined as follows.

$$\Pr\left[\begin{array}{l}(sk, pk) \xleftarrow{\$} \mathsf{KeyGen}(1^k);\\ (m, s) \xleftarrow{\$} \mathcal{A}(pk)^{\leftrightarrow \mathcal{O}_s(sk,\cdot)} :\\ 1 = \mathsf{VfSig}(pk, s, m) \wedge\\ m \in \mathcal{M} \ \wedge\ m \notin S_s\end{array}\right] \leq \epsilon(k)$$

3.3 Commitment schemes

A commitment scheme consists of algorithms CSetup, Com and VfCom. The algorithm $\mathsf{CSetup}(1^k)$ generates the parameters of the commitment scheme par_c, which include a description of the message space $\mathcal{M}$. $\mathsf{Com}(par_c, x)$ outputs a commitment com to $x \in \mathcal{M}$ and some auxiliary information $open$. The verification algorithm $\mathsf{VfCom}(par_c, com, x, open)$ outputs 1 if com is a commitment to $x \in \mathcal{M}$ with some auxiliary information $open$ or 0 if that is not the case.

Definition 3 A commitment scheme should fulfill the following correctness, hiding and binding properties.

Correctness Correctness requires that VfCom accepts all commitments created by the algorithm Com, i.e., for all $x \in \mathcal{M}$

$$\Pr\left[\begin{array}{c}par_c \xleftarrow{\$} \mathsf{CSetup}(1^k);\\ (com, open) \xleftarrow{\$} \mathsf{Com}(par_c, x) :\\ 1 = \mathsf{VfCom}(par_c, com, x, open)\end{array}\right] = 1.$$

Hiding The hiding property ensures that a commitment com to x does not reveal any information about x. For any PPT adversary $\mathcal{A}$, the hiding property is defined as follows:

$$\Pr\left[\begin{array}{l}par_c \xleftarrow{\$} \mathsf{CSetup}(1^k);\\ (x_0, st) \xleftarrow{\$} \mathcal{A}(par_c);\\ x_1 \xleftarrow{\$} \mathcal{M};\\ b \xleftarrow{\$} \{0, 1\};\\ (com, open) \xleftarrow{\$} \mathsf{Com}(par_c, x_b);\\ b' \xleftarrow{\$} \mathcal{A}(st, com) :\\ x_0 \in \mathcal{M} \ \wedge\ b = b'\end{array}\right] \leq \frac{1}{2} + \epsilon(k).$$

Binding The binding property ensures that com cannot be opened to another value x'. For any PPT adversary $\mathcal{A}$, the binding property is defined as follows:

$$\Pr\left[\begin{array}{l}par_c \xleftarrow{\$} \mathsf{CSetup}(1^k);\\ (com, x, open, x', open') \xleftarrow{\$} \mathcal{A}(par_c) :\\ x \in \mathcal{M} \ \wedge\ x' \in \mathcal{M} \ \wedge x \neq x' \wedge\\ 1 = \mathsf{VfCom}(par_c, com, x, open) \wedge\\ 1 = \mathsf{VfCom}(par_c, com, x', open')\end{array}\right] \leq \epsilon(k).$$

3.4 Polynomial commitments

A polynomial commitment scheme [29] consists of the following algorithms.

$\mathsf{PSetup}(1^k, \ell)$ On input the security parameter 1^k and an upper bound for the polynomial degree ℓ, output the parameters par_p, which include a description of the polynomial space $\mathcal{M}$.
$\mathsf{PCommit}(par_p, \phi(x))$ On input the parameters par_p and a polynomial $\phi(x) \in \mathcal{M}$, output a commitment C to $\phi(x)$ and decommitment information d.
$\mathsf{PProve}(par_p, \phi(x), i, d)$ Output a witness w that $\phi(i)$ is the evaluation of $\phi(x)$ on input i.
$\mathsf{PVerify}(par_p, C, i, \phi(i), w)$ Output 1 if w is a valid witness that $\phi(i)$ is the evaluation of $\phi(x)$ on input i. Otherwise, output 0.

Definition 4 A polynomial commitment scheme should fulfill the correctness and evaluation binding properties.

Correctness Correctness ensures that the output of PProve is always accepted by $\mathsf{PVerify}$. More formally, for all $\phi(x) \in \mathcal{M}$:

$$\Pr\left[\begin{array}{l} par_p \xleftarrow{\$} \mathsf{PSetup}(1^k, \ell); \\ (C, d) \xleftarrow{\$} \mathsf{PCommit}(par_p, \phi(x)); \\ w \leftarrow \mathsf{PProve}(par_p, \phi(x), i, d) : \\ 1 = \mathsf{PVerify}(par_p, C, i, \phi(i), w) \end{array}\right] = 1$$

Evaluation Binding A commitment to a polynomial $\phi(x)$ cannot be opened to two different evaluations $\phi(i)$ and $\phi(i)'$ on input i. More formally, for any ppt adversary $\mathcal{A}$, the evaluation binding property is defined as follows.

$$\Pr\left[\begin{array}{l} par_p \xleftarrow{\$} \mathsf{PSetup}(1^k, \ell); \\ (C, i, \langle\phi(i), w\rangle, \langle\phi(i)', w'\rangle) \xleftarrow{\$} \mathcal{A}(par_p) : \\ 1 = \mathsf{PVerify}(par_p, C, i, \phi(i), w) \wedge \\ 1 = \mathsf{PVerify}(par_p, C, i, \phi(i)', w') \wedge \\ \phi(i) \neq \phi(i)' \end{array}\right] \leq \epsilon(k)$$

4 Construction of privacy-preserving billing

We describe our construction for privacy-preserving billing. Construction BIL involves a provider $\mathcal{V}$, users $\mathcal{U}_i$, meters $\mathcal{M}_j$ and verifying parties $\mathcal{P}$.

First, we provide a generic description of construction BIL in Figs. 4 and 5. This description does not depend on the type of tariff policy being used. In Sects. 4.1, 4.2 and 4.3, we give the details of our construction when, respectively, a linear policy, a cumulative policy and a polynomial policy are employed. In Sect. 4.4, we discuss other policies.

Construction BIL is parameterized by a universe of policies U_y, a universe of consumptions U_c, a universe of times U_t, a universe of billing periods U_{bp} and a maximum size M_{max} for the meter lists. We denote by $\mathcal{U}$ the universe of user identities and by $\mathcal{M}$ the universe of meter identities.

Construction BIL uses a commitment scheme (CSetup, Com, VfCom). The provider employs a signature scheme (KeyGen_1, Sign_1, VfSig_1) to sign tariff policies, whose message space is specific to each of the tariff policies, and a signature scheme (KeyGen_2, Sign_2, VfSig_2) to sign meter lists, whose message space is $(U_{bp}, \mathcal{U}, \mathcal{M}^{M_{max}})$. The meters employ a signature scheme (KeyGen_3, Sign_3, VfSig_3) to sign meter readings, whose message space is $(\mathcal{U}, U_{bp}, \mathbb{N}, U_c, U_t)$, and a signature scheme (KeyGen_4, Sign_4, VfSig_4) to sign the number of meter readings in a billing period, whose message space is $(\mathcal{U}, U_{bp}, \mathbb{N})$. Construction BIL also employs a NIPK scheme ($\mathsf{PKSetup}$, $\mathsf{PKProve}$, $\mathsf{PKVerify}$) for a relation R. The relation R is specific to each of the tariff policies. Construction BIL works in the $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{REG}}^{\text{REG.Ver}}$ and $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$-hybrid model. $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{REG}}^{\text{REG.Ver}}$ and $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$ are described in Sect. 2.1.

When a polynomial tariff policy is employed, construction BIL also employs a polynomial commitment scheme (PSetup, $\mathsf{PCommit}$, PProve, $\mathsf{PVerify}$). In our generic description of the construction, we employ the box $\boxed{\text{POL: } \ldots}$ to denote computations that only occur when a polynomial policy is used.

The provider $\mathcal{V}$, users $\mathcal{U}_i$, meters $\mathcal{M}_j$ and verifying parties $\mathcal{P}$ are activated through the $\mathsf{bil.policy}.*$, $\mathsf{bil.listmeters}.*$, $\mathsf{bil.consumption}.*$, $\mathsf{bil.period}.*$ and $\mathsf{bil.payment}.*$ interfaces. We describe on a high level the computations performed for each of these interfaces.

1. The provider $\mathcal{V}$ receives ($\mathsf{bil.policy.ini}$, sid, bp, Y) as input. If the parameters of the scheme are not stored, $\mathcal{V}$ gets the parameters of the commitment scheme and of the NIPK scheme from $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$. In the case of a polynomial policy, $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$ also provides the parameters of the polynomial commitment scheme. If the signing key is not stored, $\mathcal{V}$ also creates a key pair for the signature scheme that signs the tariff policies. Next, $\mathcal{V}$ proceeds to sign the tariff policy. The concrete method to sign the tariff policy is described in Sects. 4.1, 4.2 and 4.3 for the linear, cumulative and polynomial policies. Finally, $\mathcal{V}$ registers the signing public key and the signed tariff policy with a new instance of $\mathcal{F}_{\text{REG}}^{\text{REG.Ver}}$ for the billing period bp.
2. The provider $\mathcal{V}$ receives ($\mathsf{bil.listmeters.ini}$, sid, bp, $\mathcal{U}_i$, $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$) as input. If a list of meters for the user $\mathcal{U}_i$ at the billing period bp was already sent, $\mathcal{V}$ aborts. Else, if the signing key is not stored, $\mathcal{V}$ creates a key pair for the signature scheme that signs the lists of meters and registers the public key with $\mathcal{F}_{\text{REG}}$. $\mathcal{V}$ signs the list of meters $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$ and sends the list of meters and the signature to the user $\mathcal{U}_i$ through an instance of the functionality $\mathcal{F}_{\text{SMT}}$. $\mathcal{U}_i$ aborts if a list of meters for the

Construction BIL**: Interfaces** bil.policy.∗**,** bil.listmeters.∗ **and** bil.consumption.∗

Construction BIL involves a provider $\mathcal{V}$, users $\mathcal{U}_i$, meters $\mathcal{M}_j$ and verifying parties $\mathcal{P}$. We denote by $\mathcal{U}$ the universe of user identities and by $\mathcal{M}$ the universe of meter identities. Construction BIL is parameterized by a security parameter 1^k. It is also parameterized by a universe of policies U_y, a universe of consumptions U_c, a universe of times U_t, a universe of billing periods U_{bp}, and a maximum size M_{max} for the meter lists. Construction BIL uses a commitment scheme (CSetup, Com, VfCom) and a NIPK scheme (PKSetup, PKProve, PKVerify). The provider employs a signature scheme (KeyGen_1, Sign_1, VfSig_1), whose message space is specific to each of the tariff policies, and another signature scheme (KeyGen_2, Sign_2, VfSig_2), whose message space is $(U_{bp}, \mathcal{U}, \mathcal{M}^{M_{max}})$. The meters employ a signature scheme (KeyGen_3, Sign_3, VfSig_3), whose message space is $(\mathcal{U}, U_{bp}, \{0,1\}^L, U_c, U_t)$ (L is large enough to avoid collisions), and (KeyGen_4, Sign_4, VfSig_4), whose message space is $(\mathcal{U}, U_{bp}, \mathbb{N})$. Construction BIL employs a polynomial commitment scheme (PSetup, PCommit, PProve, PVerify). Construction BIL works in the $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{REG}}$, $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$ and $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$-hybrid model, where CRS.Setup consists of the algorithms (PSetup, CSetup, PKSetup).

1. On input (bil.policy.ini, sid, bp, Y), $\mathcal{V}$ does the following:
   - Abort if $sid \neq (\mathcal{V}, sid')$, or if $bp \notin U_{bp}$, or if $Y \notin U_y$.
   - If the parameters POL: par_p, par_c and par_{pk} are not stored, send the message (crs.get.ini, sid) to $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$, receive the message (crs.get.end, sid, $\langle$ POL: par_p, $par_c, par_{pk}\rangle$) from the functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$, and store POL: par_p, par_c and par_{pk}. The functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ runs POL: $par_p \leftarrow \mathsf{PSetup}(1^k, \ell)$ (ℓ is the maximum degree of the polynomials in the policy), $par_c \leftarrow \mathsf{CSetup}(1^k)$ and $par_{pk} \leftarrow \mathsf{PKSetup}(1^k)$.
   - If (sk_1, pk_1) is not stored, run $(sk_1, pk_1) \leftarrow \mathsf{KeyGen}_1(1^k)$ and store (sk_1, pk_1).
   - Compute a signed tariff policy Y_s as described in Section 4.1, Section 4.2 or Section 4.3.
   - Send (reg.register.ini, $\langle sid, bp\rangle$, $\langle pk_1, Y_s\rangle$) to $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$ and receive (reg.register.end, $\langle sid, bp\rangle$) from $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$. In Section 4.1, Section 4.2 or Section 4.3, we describe REG.Ver.
   - Output (bil.policy.end, sid).
2. On input (bil.listmeters.ini, sid, bp, $\mathcal{U}_i$, $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$), $\mathcal{V}$ and $\mathcal{U}_i$ do the following:
   - $\mathcal{V}$ aborts if $bp \notin U_{bp}$, or if $\mathcal{U}_i$ is not a user identifier, or if $m > M_{max}$, or if, for $k = 1$ to m, $\mathcal{M}_{j_m}$ is not a meter identifier.
   - $\mathcal{V}$ aborts if $(sid, bp', \mathcal{U}_i', \mathcal{M}_{j_1}', \ldots, \mathcal{M}_{j_m}', s)$ such that $bp = bp'$ and $\mathcal{U}_i = \mathcal{U}_i'$ is already stored.
   - If (sk_2, pk_2) is not stored, run $(sk_2, pk_2) \leftarrow \mathsf{KeyGen}_2(1^k)$, send (reg.register.ini, sid, pk_2) to $\mathcal{F}_{\mathrm{REG}}$, receive (reg.register.end, sid, pk_2) from $\mathcal{F}_{\mathrm{REG}}$ and store (sk_2, pk_2).
   - $\mathcal{V}$ stores $(sid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m})$.
   - $\mathcal{V}$ signs $s \leftarrow \mathsf{Sign}_2(sk_2, \langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}\rangle)$.
   - $\mathcal{V}$ sets $sid_{\mathrm{SMT}} \leftarrow (\mathcal{V}, \mathcal{U}_i, sid)$ and sends (smt.send.ini, sid_{SMT}, $\langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s\rangle$) to $\mathcal{F}_{\mathrm{SMT}}$.
   - $\mathcal{U}_i$ receives (smt.send.end, sid_{SMT}, $\langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s\rangle$) from $\mathcal{F}_{\mathrm{SMT}}$.
   - $\mathcal{U}_i$ aborts if $(sid, bp', \mathcal{M}_{j_1}', \ldots, \mathcal{M}_{j_m}', s)$ such that $bp = bp'$ is already stored.
   - If pk_2 is not stored, $\mathcal{U}_i$ sends (reg.retrieve.ini, sid) to $\mathcal{F}_{\mathrm{REG}}$, receives (reg.retrieve.end, sid, pk_2), and stores pk_2.
   - $\mathcal{U}_i$ runs $b \leftarrow \mathsf{VfSig}_2(pk_2, s, \langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}\rangle)$.
   - $\mathcal{U}_i$ aborts if $b = 0$.
   - $\mathcal{U}_i$ stores $(sid, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s)$.
   - $\mathcal{U}_i$ outputs (bil.listmeters.end, sid, bp, $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$).
3. On input (bil.consumption.ini, $sid, \mathcal{U}_i$, bp, c, t), $\mathcal{M}_j$ and $\mathcal{U}_i$ do the following:
   - $\mathcal{M}_j$ aborts if $\mathcal{U}_i$ is not a user identifier, or if $bp \notin U_{bp}$, or if $c \notin U_c$, or if $t \notin U_t$.
   - $\mathcal{M}_j$ aborts if $(sid, \mathcal{U}_i', bp', ctm[bp, \mathcal{U}_i], s)$ such that $\mathcal{U}_i' = \mathcal{U}_i$ and $bp' = bp$ is already stored.
   - If $(sk_{3,k}, pk_{3,k})$ and $(sk_{4,k}, pk_{4,k})$ are not stored, $\mathcal{M}_j$ runs $(sk_{3,k}, pk_{3,k}) \leftarrow \mathsf{KeyGen}_3(1^k)$ and $(sk_{4,k}, pk_{4,k}) \leftarrow \mathsf{KeyGen}_4(1^k)$, sends (reg.register.ini, $\langle sid, \mathcal{M}_j\rangle$, $\langle pk_{3,k}, pk_{4,k}\rangle$) to $\mathcal{F}_{\mathrm{REG}}$, receives (reg.register.end, $\langle sid, \mathcal{M}_j\rangle$, $\langle pk_{3,k}, pk_{4,k}\rangle$) from $\mathcal{F}_{\mathrm{REG}}$ and stores $(sk_{3,k}, pk_{3,k})$ and $(sk_{4,k}, pk_{4,k})$.
   - $\mathcal{M}_j$ increments a counter $ctm[bp, \mathcal{U}_i]$ (initialized at zero).
   - $\mathcal{M}_j$ runs $s \leftarrow \mathsf{Sign}_3(sk_{3,k}, \langle \mathcal{U}_i, bp, ctm[bp, \mathcal{U}_i], c, t\rangle)$.
   - $\mathcal{M}_j$ sets $sid_{\mathrm{SMT}} \leftarrow (\mathcal{M}_j, \mathcal{U}_i, sid)$ and sends (smt.send.ini, sid_{SMT}, $\langle \mathcal{U}_i, bp, ctm[bp, \mathcal{U}_i], c, t, s\rangle$) to $\mathcal{F}_{\mathrm{SMT}}$.
   - $\mathcal{U}_i$ receives (smt.send.end, sid_{SMT}, $\langle \mathcal{U}_i, bp, ctm[bp, \mathcal{M}_j], c, t, s\rangle$) from $\mathcal{F}_{\mathrm{SMT}}$.
   - $\mathcal{U}_i$ aborts if $(sid, \mathcal{M}_j', bp', ctm'[bp, \mathcal{M}_j], s')$ such that $\mathcal{M}_j' = \mathcal{M}_j$ and $bp' = bp$ is already stored.
   - If $pk_{3,k}$ and $pk_{4,k}$ are not stored, $\mathcal{U}_i$ sends (reg.retrieve.ini, $\langle sid, \mathcal{M}_j\rangle$) to $\mathcal{F}_{\mathrm{REG}}$, receives (reg.retrieve.end, $\langle sid, \mathcal{M}_j\rangle$, $\langle pk_{3,k}, pk_{4,k}\rangle$), and stores $pk_{3,k}$ and $pk_{4,k}$.
   - $\mathcal{U}_i$ runs $b \leftarrow \mathsf{VfSig}_3(pk_{3,k}, s, \langle \mathcal{U}_i, bp, ctm[bp, \mathcal{M}_j], c, t\rangle)$.
   - $\mathcal{U}_i$ aborts if $b = 0$.
   - For all the tuples $[\mathcal{M}_j', bp', ctm'[bp, \mathcal{M}_j], c, t, s]$ stored such that $\mathcal{M}_j' = \mathcal{M}_j$ and $bp' = bp$, $\mathcal{U}_i$ aborts if $ctm'[bp, \mathcal{M}_j] = ctm[bp, \mathcal{M}_j]$.
   - $\mathcal{U}_i$ stores $[\mathcal{M}_j, bp, ctm[bp, \mathcal{M}_j], c, t, s]$.
   - $\mathcal{U}_i$ outputs (bil.consumption.end, sid, $\mathcal{M}_j$, bp, c, t).

Fig. 4 Construction BIL: interfaces bil.policy.∗, bil.listmeters.∗ and bil.consumption.∗

Construction BIL**: Interfaces** bil.period.∗ **and** bil.payment.∗

4. On input $(\mathsf{bil.period.ini}, sid, \mathcal{U}_i, bp)$, $\mathcal{M}_j$ and $\mathcal{U}_i$ do the following:
   - $\mathcal{M}_j$ aborts if $\mathcal{U}_i$ is not a user identifier, or if $bp \notin U_{bp}$.
   - $\mathcal{M}_j$ aborts if $(sid, \mathcal{U}_i', bp', ctm[bp, \mathcal{U}_i])$ such that $\mathcal{U}_i' = \mathcal{U}_i$ and $bp' = bp$ is already stored.
   - $\mathcal{M}_j$ stores $(sid, \mathcal{U}_i, bp, ctm[bp, \mathcal{U}_i])$. The counter $ctm[bp, \mathcal{U}_i]$ equals 0 if $\mathcal{M}_j$ did not send any meter reading to $\mathcal{U}_i$ at the billing period bp.
   - If $(sk_{3,k}, pk_{3,k})$ and $(sk_{4,k}, pk_{4,k})$ are not stored, $\mathcal{M}_j$ runs $(sk_{3,k}, pk_{3,k}) \leftarrow \mathsf{KeyGen_3}(1^k)$ and $(sk_{4,k}, pk_{4,k}) \leftarrow \mathsf{KeyGen_4}(1^k)$, sends $(\mathsf{reg.register.ini}, \langle sid, \mathcal{M}_j \rangle, \langle pk_{3,k}, pk_{4,k} \rangle)$ to $\mathcal{F}_{\mathrm{REG}}$, receives $(\mathsf{reg.register.end}, \langle sid, \mathcal{M}_j \rangle)$ from $\mathcal{F}_{\mathrm{REG}}$ and stores $(sk_{3,k}, pk_{3,k})$ and $(sk_{4,k}, pk_{4,k})$.
   - $\mathcal{M}_j$ runs $s \leftarrow \mathsf{Sign_4}(sk_{4,k}, \langle \mathcal{U}_i, bp, ctm[bp, \mathcal{U}_i] \rangle)$.
   - $\mathcal{M}_j$ sets $sid_{\mathrm{SMT}} \leftarrow (\mathcal{M}_j, \mathcal{U}_i, sid)$ and sends $(\mathsf{smt.send.ini}, sid_{\mathrm{SMT}}, \langle \mathcal{U}_i, bp, ctm[bp, \mathcal{U}_i], s \rangle)$ to $\mathcal{F}_{\mathrm{SMT}}$.
   - $\mathcal{U}_i$ receives $(\mathsf{smt.send.end}, sid_{\mathrm{SMT}}, \langle \mathcal{U}_i, bp, ctm[bp, \mathcal{M}_j], s \rangle)$ from $\mathcal{F}_{\mathrm{SMT}}$.
   - $\mathcal{U}_i$ aborts if there is a tuple $(sid, \mathcal{M}_j', bp', ctm'[bp, \mathcal{M}_j], s)$ stored such that $\mathcal{M}_j' = \mathcal{M}_j$ and $bp' = bp$.
   - If $pk_{3,k}$ and $pk_{4,k}$ are not stored, $\mathcal{U}_i$ sends $(\mathsf{reg.retrieve.ini}, \langle sid, \mathcal{M}_j \rangle)$ to $\mathcal{F}_{\mathrm{REG}}$, receives $(\mathsf{reg.retrieve.end}, \langle sid, \mathcal{M}_j \rangle, \langle pk_{3,k}, pk_{4,k} \rangle)$, and stores $pk_{3,k}$ and $pk_{4,k}$.
   - $\mathcal{U}_i$ runs $b \leftarrow \mathsf{VfSig_4}(pk_{4,k}, s, \langle \mathcal{U}_i, bp, ctm[bp, \mathcal{M}_j] \rangle)$.
   - $\mathcal{U}_i$ aborts if $b = 0$.
   - $\mathcal{U}_i$ counts the number of tuples $[\mathcal{M}_j', bp', ctm'[bp, \mathcal{M}_j], c, t, s]$ stored such that $\mathcal{M}_j' = \mathcal{M}_j$ and $bp' = bp$. If the number is different from $ctm[bp, \mathcal{M}_j]$, $\mathcal{U}_i$ aborts. $\mathcal{U}_i$ also aborts if, from $d = 1$ to $ctm[bp, \mathcal{M}_j]$, $\mathcal{U}_i$ cannot find a tuple $[\mathcal{M}_j', bp', ctm'[bp, \mathcal{M}_j], c, t, s]$ stored such that $\mathcal{M}_j' = \mathcal{M}_j$ and $bp' = bp$ and $d = ctm'[bp, \mathcal{M}_j]$.
   - $\mathcal{U}_i$ stores $(sid, \mathcal{M}_j, bp, ctm[bp, \mathcal{M}_j], s)$.
   - $\mathcal{U}_i$ outputs $(\mathsf{bil.period.end}, sid, bp, \mathcal{M}_j, ctm[bp, \mathcal{M}_j])$.
5. On input $(\mathsf{bil.payment.ini}, sid, \mathcal{P}, bp)$, $\mathcal{U}_i$ and $\mathcal{P}$ do the following:
   - $\mathcal{U}_i$ aborts if $\mathcal{P}$ is not a valid party identifier, of if $bp \notin U_{bp}$.
   - If $\boxed{\text{POL: } par_p,}$ par_c and par_{pk} are not stored, $\mathcal{U}_i$ sends the message $(\mathsf{crs.get.ini}, sid)$ to $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$, receives the message $(\mathsf{crs.get.end}, sid, \langle \boxed{\text{POL: } par_p,} par_c, par_{pk} \rangle)$ from $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$, and stores $\boxed{\text{POL: } par_p,}$ par_c and par_{pk}.
   - If $(sid, bp', \langle pk_1, Y_s \rangle)$ such that $bp = bp'$ is not stored, $\mathcal{U}_i$ sends $(\mathsf{reg.retrieve.ini}, \langle sid, bp \rangle)$ to $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$, receives $(\mathsf{reg.retrieve.end}, \langle sid, bp \rangle, \langle pk_1, Y_s \rangle)$ from $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$, and stores $(sid, bp, \langle pk_1, Y_s \rangle)$. $\mathcal{U}_i$ aborts if $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$ sends $(\mathsf{reg.retrieve.end}, \langle sid, bp \rangle, \perp)$.
   - If a meter list $(sid, bp', \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s)$ such that $bp' = bp$ is not stored, $\mathcal{U}_i$ aborts.
   - For $k = 1$ to m, $\mathcal{U}_i$ does the following:
     - Abort if a tuple $(sid, \mathcal{M}_j', bp', ctm[bp, \mathcal{M}_j])$ such that $\mathcal{M}_{j_k} = \mathcal{M}_j'$ and $bp' = bp$ is not stored.
     - Set $p_k = 0$.
     - For $d = 1$ to $ctm[bp, \mathcal{M}_j]$, retrieve each of the $ctm[bp, \mathcal{M}_j]$ tuples $[\mathcal{M}_j', bp', d, c_d, t_d, s_d]$ such that $\mathcal{M}_{j_k} = \mathcal{M}_j'$ and $bp' = bp$ and set $p_k = p_k + Y(c_d, t_d)$.
   - Set $p = \sum_{k=1}^{m} p_k$.
   - Run $(com, open) \leftarrow \mathsf{Com}(par_c, p)$.
   - Set wit and ins for a relation R as described in Section 4.1, Section 4.2 or Section 4.3.
   - Run $\pi \leftarrow \mathsf{PKProve}(par_{pk}, wit, ins)$.
   - $\mathcal{U}_i$ sets $sid_{\mathrm{SMT}} \leftarrow (\mathcal{U}_i, \mathcal{P}, sid)$ and sends $(\mathsf{smt.send.ini}, sid_{\mathrm{SMT}}, \langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi \rangle)$ to $\mathcal{F}_{\mathrm{SMT}}$.
   - $\mathcal{P}$ receives $(\mathsf{smt.send.end}, sid_{\mathrm{SMT}}, \langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi \rangle)$ from $\mathcal{F}_{\mathrm{SMT}}$.
   - If $(sid, bp', \langle pk_1, Y_s \rangle)$ such that $bp = bp'$ is not stored, $\mathcal{U}_i$ sends $(\mathsf{reg.retrieve.ini}, \langle sid, bp \rangle)$ to $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$, receives $(\mathsf{reg.retrieve.end}, \langle sid, bp \rangle, \langle pk_1, Y_s \rangle)$ from $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$, and stores $(sid, bp, \langle pk_1, Y_s \rangle)$. $\mathcal{U}_i$ aborts if $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$ sends $(\mathsf{reg.retrieve.end}, \langle sid, bp \rangle, \perp)$.
   - If pk_2 is not stored, $\mathcal{P}$ sends $(\mathsf{reg.retrieve.ini}, sid)$ to $\mathcal{F}_{\mathrm{REG}}$, receives $(\mathsf{reg.retrieve.end}, sid, pk_2)$ from $\mathcal{F}_{\mathrm{REG}}$, and stores pk_2.
   - $\mathcal{P}$ aborts if $1 \neq \mathsf{VfSig_2}(pk_2, s, \langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m} \rangle)$.
   - For $k = 1$ to m, If $pk_{3,k}$ and $pk_{4,k}$ are not stored, $\mathcal{P}$ sends $(\mathsf{reg.retrieve.ini}, \langle sid, \mathcal{M}_j \rangle)$ to $\mathcal{F}_{\mathrm{REG}}$, receives $(\mathsf{reg.retrieve.end}, \langle sid, \mathcal{M}_{j_k} \rangle, \langle pk_{3,k}, pk_{4,k} \rangle)$, and stores $pk_{3,k}$ and $pk_{4,k}$.
   - If $\boxed{\text{POL: } par_p,}$ par_c and par_{pk} are not stored, $\mathcal{P}$ sends the message $(\mathsf{crs.get.ini}, sid)$ to $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$, receives the message $(\mathsf{crs.get.end}, sid, \langle \boxed{\text{POL: } par_p,} par_c, par_{pk} \rangle)$ from $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$, and stores $\boxed{\text{POL: } par_p,}$ par_c and par_{pk}.
   - $\mathcal{P}$ checks that the instance ins is consistent with the received values pk_1, $pk_{3,k}$, $pk_{4,k}$, $\boxed{\text{POL: } par_p,}$ par_c and par_{pk}. $\mathcal{P}$ also checks that, for $k = 1$ to m, the instance includes a counter $ctm[bp, \mathcal{M}_{j_k}]$ of meter readings and that the proof proves possession of $ctm[bp, \mathcal{M}_{j_k}]$ meter readings numbered from 1 to $ctm[bp, \mathcal{M}_{j_k}]$.
   - $\mathcal{P}$ aborts if $1 \neq \mathsf{PKVerify}(par_{pk}, ins, \pi)$.
   - $\mathcal{P}$ aborts if $1 \neq \mathsf{VfCom}(par_c, com, p, open)$.
   - $\mathcal{P}$ retrieves $(ctm[bp, \mathcal{M}_{j_1}], \ldots, ctm[bp, \mathcal{M}_{j_m}])$ from ins.
   - $\mathcal{P}$ outputs $(\mathsf{bil.payment.end}, sid, \mathcal{U}_i, bp, p, \mathcal{M}_{j_1}, ctm[bp, \mathcal{M}_{j_1}], \ldots, \mathcal{M}_{j_m}, ctm[bp, \mathcal{M}_{j_m}])$.

Fig. 5 Construction BIL: interfaces bil.period.∗ and bil.payment.∗

billing period bp was received before or if the signature is not correct, else $\mathcal{U}_i$ outputs the list of meters.

3. The meter $\mathcal{M}_j$ receives (bil.consumption.ini, $sid, \mathcal{U}_i$, bp, c, t) as input. $\mathcal{M}_j$ aborts if the end of billing period message was already sent to $\mathcal{U}_i$ at the billing period bp. If the signing keys are not stored, $\mathcal{M}_j$ creates key pairs for the signature schemes that sign meter readings and the number of meter readings output in a billing period, and registers them with an instance of $\mathcal{F}_{\text{REG}}$. Next, $\mathcal{M}_j$ increments a counter $ctm[bp, \mathcal{U}_i]$ that counts the number of meter readings sent to the user $\mathcal{U}_i$ during the billing period bp. $\mathcal{M}_j$ signs the meter reading $(\mathcal{U}_i, bp, ctm[bp, \mathcal{U}_i], c, t)$ and sends the meter reading and the signature to the user $\mathcal{U}_i$ through an instance of the functionality $\mathcal{F}_{\text{SMT}}$. $\mathcal{U}_i$ aborts if the end of billing period message was already received from the meter $\mathcal{M}_j$ at the billing period bp. If $\mathcal{U}_i$ does not store the signing public key, $\mathcal{U}_i$ retrieves it from the instance of $\mathcal{F}_{\text{REG}}$. $\mathcal{U}_i$ verifies the signature, checks that the counter value in the meter reading does not equal the counter value of any of the stored meter readings from that meter at that billing period, and outputs the meter reading.
4. The meter $\mathcal{M}_j$ receives (bil.period.ini, $sid, \mathcal{U}_i, bp$) as input. $\mathcal{M}_j$ aborts if the end of period message was already sent to $\mathcal{U}_i$ at the billing period bp. $\mathcal{M}_j$ signs the counter that counts the number of meter readings sent to the user $\mathcal{U}_i$ during the billing period bp. $\mathcal{M}_j$ sends the counter and the signature to $\mathcal{U}_i$ through an instance of the functionality $\mathcal{F}_{\text{SMT}}$. $\mathcal{U}_i$ aborts if the end of period message was already received from $\mathcal{M}_j$ at the billing period bp. $\mathcal{U}_i$ verifies the signature, checks that it stores the number of meter readings indicated by the counter and that the meter readings are numbered from 1 to the counter value, and outputs the billing period bp, the meter identity $\mathcal{M}_j$ and the number of meter readings.
5. The meter $\mathcal{U}_i$ receives (bil.payment.ini, $sid, \mathcal{P}, bp$) as input. If the parameters of the scheme are not stored, $\mathcal{U}_i$ gets them from $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$. If the public key and the signed policy are not stored, $\mathcal{U}_i$ retrieves them from the corresponding instance of $\mathcal{F}_{\text{REG}}^{\text{REG.Ver}}$. Next, $\mathcal{U}_i$ checks that $\mathcal{V}$ sent the list of meters $(\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m})$ for the billing period bp. $\mathcal{U}_i$ also checks that each of the meters in that list sent the end of billing period message. In that case, $\mathcal{U}_i$ computes the bill to be paid by applying the tariff policy to each of the meter readings and summing up the prices to be paid for each of the meter readings. $\mathcal{U}_i$ computes a non-interactive zero-knowledge proof that the bill is correctly calculated. The details for this proof are given in Sects. 4.1, 4.2 and 4.3 for the linear, cumulative and polynomial policies, respectively. $\mathcal{U}_i$ sends the signed list of meters, the bill and the proof to the verifying party $\mathcal{P}$ through an instance of the functionality $\mathcal{F}_{\text{SMT}}$. If the parameters of the scheme are not stored, $\mathcal{P}$ gets them from $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$. $\mathcal{P}$ also retrieves the public keys of the provider and of the meters. $\mathcal{P}$ verifies the signed meter list and the proof and then outputs the bill.

4.1 Linear policies

A linear policy is a tariff policy in which the time is divided into time intervals $[t_1, t_2), [t_2, t_3), \ldots [t_L, t_{L+1})$. The tariff policy associates each time interval to a rate r. The rate denotes a price per unit of consumption. The policy can be expressed as follows:

$$Y(c,t) = \left\{ \begin{array}{ll} c \cdot r_1 & \text{if } t \in [t_1, t_2) \\ \vdots & \vdots \\ c \cdot r_L & \text{if } t \in [t_L, t_{L+1}) \end{array} \right\}$$

In order to sign this tariff policy using a key pair (pk_1, sk_1) for the signature scheme $(\mathsf{KeyGen}_1, \mathsf{Sign}_1, \mathsf{VfSig}_1)$, the provider $\mathcal{V}$ proceeds as follows. For each time interval $[t_{min}, t_{max})$ in the tariff policy, $\mathcal{V}$ computes a signature $s \leftarrow \mathsf{Sign}_1(sk_1, \langle bp, r, t_{min}, t_{max}\rangle)$. The signed tariff policy Y_s consists of L tuples $[r_l, t_{min,l}, t_{max,l}, s_l]_{l=1}^{L}$. The verification function REG.Ver verifies the signatures in the signed tariff policy.

In order to compute a non-interactive zero-knowledge proof of correctness of the bill computation, $\mathcal{U}_i$ proceeds as follows. Let $(sid, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s)$ be the list of meters signed by $\mathcal{V}$. For $k = 1$ to m, let $(sid, \mathcal{M}_{j_k}, bp, ctm[bp, \mathcal{M}_{j_k}], s_k)$ be the number of meter readings sent by meter $\mathcal{M}_{j_k}$. For $d = 1$ to $ctm[bp, \mathcal{M}_{j_k}]$, let $[\mathcal{M}_{j_k}, bp, d, c_{k,d}, t_{k,d}, s_{k,d}]$ be the meter readings sent to $\mathcal{U}_i$ by $\mathcal{M}_{j_k}$. $\mathcal{U}_i$ computes a non-interactive zero-knowledge proof of knowledge for the following relation.

$$\begin{aligned}
&R = \{(ins, wit) : \\
&\{1 = \mathsf{VfSig}_4(pk_{4,k}, s_k, \langle bp, \mathcal{U}_i, ctm[bp, \mathcal{M}_{j_k}]\rangle) \ \wedge && (1)\\
&[1 = \mathsf{VfSig}_3(pk_{3,k}, s_{k,d}, \langle \mathcal{U}_i, bp, d, c_{k,d}, t_{k,d}\rangle) \ \wedge && (2)\\
&1 = \mathsf{VfSig}_1(pk_1, s'_{k,d}, \langle bp, r_{k,d}, t_{min,k,d}, t_{max,k,d}\rangle) \ \wedge && (3)\\
&t_{k,d} \in [t_{min,k,d}, t_{max,k,d}] \ \wedge && (4)\\
&p_{k,d} = c_{k,d} \cdot r_{k,d}]_{d=1}^{ctm[bp,\mathcal{M}_{j_k}]}\}_{k=1}^{m} \ \wedge && (5)\\
&p = \sum_{k=1}^{m} \sum_{d=1}^{ctm[bp,\mathcal{M}_{j_k}]} p_{k,d} \ \wedge && (6)\\
&1 = \mathsf{VfCom}(par_c, com, p, open)\}. && (7)
\end{aligned}$$

In this relation, for each meter $\mathcal{M}_{j_k}$ in the meter list, Line 1 requires the user to prove knowledge of the signature s_k from $\mathcal{M}_{j_k}$ on the tuple $\langle bp, \mathcal{U}_i, ctm[bp, \mathcal{M}_{j_k}]\rangle$. This signed tuple belongs to the instance of the proof. Despite the fact that the

signed values are revealed in the proof instance, the signature s_k must belong to the witness to prevent a malicious meter from disclosing information to the verifying party through s_k. For the $ctm[bp, \mathcal{M}_{j_k}]$ meter readings that $\mathcal{M}_{j_k}$ sent to $\mathcal{U}_i$, Line 2 requires the user to prove knowledge of a meter reading $c_{k,d}$ and $t_{k,d}$ and of a signature $s_{k,d}$ from $\mathcal{M}_{j_k}$ on that meter reading. The signed values $\mathcal{U}_i$, bp and d belong to the proof instance. Line 3 requires the user to prove knowledge of the rate $r_{k,d}$, of an interval $[t_{min,k,d}, t_{max,k,d})$, and of a signature $s'_{k,d}$ in the tariff policy that signs those values. The signed billing period bp belongs to the proof instance. Line 4 is a range proof that requires the user to prove that the time $t_{k,d}$ proven in Line 2 lies within the interval $[t_{min,k,d}, t_{max,k,d})$ proven in Line 3. Thanks to that, the verifier ensures that the user employs the rate $r_{k,d}$ associated with the correct time interval in the tariff policy. Line 5 requires the user to prove that the price associated with the meter reading proven in Line 2 is computed by multiplying the rate $r_{k,d}$ by the consumption $c_{k,d}$. Finally, Line 6 and Line 7 require the user to prove that com is a commitment to the total price, which is computed by summing up the prices for each meter reading.

$\mathcal{U}_i$ sets the witness as follows.

$$wit \leftarrow (p, open, [\langle c_{k,d}, t_{k,d}, p_{k,d}, s_{k,d}, s'_{k,d}, r_{k,d}, t_{min,k,d}, t_{max,k,d}\rangle_{d=1}^{ctm[bp,\mathcal{M}_{j_k}]}, s_k]_{k=1}^{m})$$

$\mathcal{U}_i$ sets the instance as follows.

$$ins \leftarrow (par_c, pk_1, \mathcal{U}_i, com, bp, [pk_{3,k}, pk_{4,k}, ctm[bp, \mathcal{M}_{j_k}]]_{k=1}^{m}).$$

The verifying party $\mathcal{P}$, in order to verify the statement $1 = \mathsf{VfSig}_3(pk_{3,k}, s_{k,d}, \langle \mathcal{U}_i, bp, d, c_{k,d}, t_{k,d}\rangle)$ in Line 2 of the relation, must employ values d from 1 to $ctm[bp, \mathcal{M}_{j_k}]$.

4.2 Cumulative policies

A cumulative policy is a tariff policy in which, as in the linear policy, the time is divided into time intervals $[t_1, t_2)$, $[t_2, t_3), \ldots [t_L, t_{L+1})$. Additionally, for each time interval $[t_l, t_{l+1})$, the consumption is also divided into intervals $[c_{l,1}, c_{l,2})$, $[c_{l,2}, c_{l,3}), \ldots [c_{l,M}, c_{l,M+1})$. The tariff policy associates with each time interval a set of rates, one for each consumption interval. The rate denotes a price per unit of consumption. The policy can be expressed as follows:

$$Y(c,t) = \left\{ \begin{array}{ll} \Phi_1(c) \text{ if } & t \in [t_1, t_2) \\ \vdots & \vdots \\ \Phi_L(c) \text{ if } & t \in [t_L, t_{L+1}) \end{array} \right\}$$

Each of functions $\Phi_l(c)$ ($l \in [1, L]$) is defined as follows.

$$\Phi_l(c) = \left\{ \begin{array}{ll} (c - c_1) \cdot r_1 + F_1 & \text{if } c \in [c_1, c_2) \\ \vdots & \vdots \\ (c - c_M) \cdot r_M + F_M & \text{if } c \in [c_M, c_{M+1}) \end{array} \right\}$$

Therefore, for a meter reading (c, t), the price to be paid is defined by the function $\Phi_l(c)$ associated with the time interval such that $t \in [t_l, t_{l+1})$. For a consumption c such that $c \in [c_m, c_{m+1})$, the function $\Phi_l(c)$ is $(c - c_m) \cdot r_m + F_m$. F_m is a constant that equals $\sum_{m'=1}^{m-1}(c_{m'+1} - c_{m'}) \cdot r_{m'}$, i.e., F_m is the price to be paid for a consumption c_m, which is computed by summing up the prices to be paid for all the previous consumption intervals.

In order to sign this tariff policy using a key pair (pk_1, sk_1) for the signature scheme $(\mathsf{KeyGen}_1, \mathsf{Sign}_1, \mathsf{VfSig}_1)$, the provider $\mathcal{V}$ proceeds as follows. For each consumption interval $[c_{min}, c_{max})$ in a function $\Phi(c)$ associated with the time interval $[t_{min}, t_{max})$, $\mathcal{V}$ computes a signature $s \leftarrow \mathsf{Sign}_1(sk_1, \langle bp, r, F, t_{min}, t_{max}, c_{min}, c_{max}\rangle)$. The signed tariff policy Y_s consists of tuples of the form $[r, F, t_{min}, t_{max}, c_{min}, c_{max}, s]$. The verification function $\mathsf{REG.Ver}$ verifies the signatures in the signed tariff policy.

In order to compute a non-interactive zero-knowledge proof of correctness of the bill computation, $\mathcal{U}_i$ computes a non-interactive zero-knowledge proof of knowledge for the following relation.

$$\begin{aligned} &R = \{(ins, wit) : \\ &\{1 = \mathsf{VfSig}_4(pk_{4,k}, s_k, \langle bp, \mathcal{U}_i, ctm[bp, \mathcal{M}_{j_k}]\rangle) \ \wedge && (8)\\ &[1 = \mathsf{VfSig}_3(pk_{3,k}, s_{k,d}, \langle \mathcal{U}_i, bp, d, c_{k,d}, t_{k,d}\rangle) \ \wedge && (9)\\ &\boxed{1 = \mathsf{VfSig}_1(pk_1, s'_{k,d}, \langle bp, r_{k,d}, F_{k,d},} \\ &\boxed{t_{min,k,d}, t_{max,k,d}, c_{min,k,d}, c_{max,k,d}\rangle) \wedge} && (10)\\ &t_{k,d} \in [t_{min,k,d}, t_{max,k,d}] \ \wedge && (11)\\ &\boxed{c_{k,d} \in [c_{min,k,d}, c_{max,k,d}] \wedge} && (12)\\ &\boxed{p_{k,d} = (c_{k,d} - c_{min,k,d}) \cdot r_{k,d} + F_{k,d}} && (13)\\ &]_{d=1}^{ctm[bp,\mathcal{M}_{j_k}]} \wedge \}_{k=1}^{m} \\ &p = \sum_{k=1}^{m} \sum_{d=1}^{ctm[bp,\mathcal{M}_{j_k}]} p_{k,d} \ \wedge && (14)\\ &1 = \mathsf{VfCom}(par_c, com, p, open)\}. && (15) \end{aligned}$$

We highlight the differences between this relation and the relation for linear policies by using boxes. Line 10 requires the user to prove knowledge of the rate $r_{k,d}$, of the constant $F_{k,d}$, of a time interval $[t_{min,k,d}, t_{max,k,d})$, of a consumption interval $[c_{min,k,d}, c_{max,k,d})$ and of a signature $s'_{k,d}$ in the tariff policy that signs those values. The signed billing period bp belongs to the proof instance. Line 12 is a range proof that requires the user to prove that the consumption $c_{k,d}$ proven in Line 9 lies within the interval $[c_{min,k,d}, c_{max,k,d})$ proven in Line 10. Thanks to that, the verifier ensures that the user employs the rate $r_{k,d}$, the value $c_{min,k,d}$, and the constant $F_{k,d}$ associated with the correct consumption interval in the tariff policy. Line 13 requires the user to prove that the price associated with the meter reading proven in Line 9 is computed by doing $(c_{k,d} - c_{min,k,d}) \cdot r_{k,d} + F_{k,d}$.

$\mathcal{U}_i$ sets the witness as follows.

$$wit \leftarrow (p, open, [\langle c_{k,d}, t_{k,d}, p_{k,d}, s_{k,d}, s'_{k,d}, r_{k,d}, \boxed{F_{k,d}}, t_{min,k,d}, t_{max,k,d} \boxed{, c_{min,k,d}, c_{max,k,d}} \rangle_{d=1}^{ctm[bp,\mathcal{M}_{j_k}]}, s_k]_{k=1}^{m})$$

$\mathcal{U}_i$ sets the instance as in the case of a linear policy. As for the linear policy, the verifying party $\mathcal{P}$, in order to verify the statement $1 = \mathsf{VfSig}_3(pk_{3,k}, s_{k,d}, \langle \mathcal{U}_i, bp, d, c_{k,d}, t_{k,d} \rangle)$ in Line 9 of the relation, must employ values d from 1 to $ctm[bp, \mathcal{M}_{j_k}]$.

4.3 Polynomial policies

A polynomial policy is a tariff policy in which, as in the cumulative policy, the time is divided into time intervals $[t_1, t_2)$, $[t_2, t_3), \ldots [t_{L-1}, t_L)$ and, for each time interval $[t_l, t_{l+1})$, the consumption is also divided into intervals $[c_{l,1}, c_{l,2})$, $[c_{l,2}, c_{l,3}), \ldots [c_{l,M}, c_{l,M+1})$. The tariff policy associates to each time interval a spline Φ. The policy can be expressed as follows:

$$Y(c,t) = \begin{Bmatrix} \Phi_1(c) & \text{if } t \in [t_1, t_2) \\ \vdots & \vdots \\ \Phi_L(c) & \text{if } t \in [t_L, t_{L+1}) \end{Bmatrix}$$

Each of the splines $\Phi_l(c)$ ($l \in [1, L]$) is defined as follows.

$$\Phi_l(c) = \begin{Bmatrix} \phi_1(c) & \text{if } c \in [c_1, c_2) \\ \vdots & \vdots \\ \phi_M(c) & \text{if } c \in [c_M, c_{M+1}) \end{Bmatrix}$$

Therefore, for a meter reading (c, t), the price to be paid is defined by the polynomial $\phi_m(c)$ such that $c \in [c_m, c_{m+1})$ that belongs to the spline $\Phi_l(c)$ associated with the time interval $[t_l, t_{l+1})$ such that $t \in [t_l, t_{l+1})$.

To compute the signed tariff policy Y_s, for all the polynomials ϕ in the tariff policy, the provider $\mathcal{V}$ computes $(C, d) \leftarrow \mathsf{PCommit}(par_p, \phi)$ and signs $s \leftarrow \mathsf{Sign}_1(sk_1, \langle bp, C, t_{min}, t_{max}, c_{min}, c_{max} \rangle)$, where $[t_{min}, t_{max})$ and $[c_{min}, c_{max})$ are the time and consumption intervals associated with the polynomial ϕ. The signed tariff policy Y_s consists of tuples of the form $[\phi, t_{min}, t_{max}, c_{min}, c_{max}, C, d, s]$. The verification function $\mathsf{REG.Ver}$ verifies the signatures in the signed tariff policy.

In order to compute a non-interactive zero-knowledge proof of correctness of the bill computation, $\mathcal{U}_i$ computes a non-interactive zero-knowledge proof of knowledge for the following relation.

$$\begin{aligned}
&R = \{(ins, wit) : \\
&\{1 = \mathsf{VfSig}_4(pk_{4,k}, s_k, \langle bp, \mathcal{U}_i, ctm[bp, \mathcal{M}_{j_k}] \rangle) \ \wedge && (16)\\
&[1 = \mathsf{VfSig}_3(pk_{3,k}, s_{k,d}, \langle \mathcal{U}_i, bp, d, c_{k,d}, t_{k,d} \rangle) \ \wedge && (17)\\
&\boxed{1 = \mathsf{VfSig}_1(pk_1, s'_{k,d}, \langle bp, C_{k,d},} \\
&\boxed{t_{min,k,d}, t_{max,k,d}, c_{min,k,d}, c_{max,k,d} \rangle)} \ \wedge && (18)\\
&t_{k,d} \in [t_{min,k,d}, t_{max,k,d}] \ \wedge && (19)\\
&c_{k,d} \in [c_{min,k,d}, c_{max,k,d}] \ \wedge && (20)\\
&\boxed{1 = \mathsf{PVerify}(par_p, C_{k,d}, c_{k,d}, p_{k,d}, w_{k,d})} && (21)\\
&]_{d=1}^{ctm[bp,\mathcal{M}_{j_k}]}\}_{k=1}^{m} \wedge \\
&p = \sum_{k=1}^{m} \sum_{d=1}^{ctm[bp,\mathcal{M}_{j_k}]} p_{k,d} \ \wedge && (22)\\
&1 = \mathsf{VfCom}(par_c, com, p, open)\}. && (23)
\end{aligned}$$

We highlight the differences between this relation and the relation for cumulative policies by using boxes. Line 18 requires the user to prove knowledge of a commitment $C_{k,d}$, of a time interval $[t_{min,k,d}, t_{max,k,d})$, of a consumption interval $[c_{min,k,d}, c_{max,k,d})$ and of a signature $s'_{k,d}$ in the tariff policy that signs those values. The signed billing period bp belongs to the proof instance. Line 21 requires the user to prove that the price $p_{k,d}$ associated with the meter reading $(c_{k,d}, t_{k,d})$ proven in Line 17 is the evaluation of the polynomial committed to in $C_{k,d}$ on input $c_{k,d}$.

$\mathcal{U}_i$ sets the witness as follows.

$$wit \leftarrow (p, open, [\langle c_{k,d}, t_{k,d}, p_{k,d}, s_{k,d}, s'_{k,d}, \boxed{C_{k,d}, w_{k,d},} t_{min,k,d}, t_{max,k,d}, c_{min,k,d}, c_{max,k,d} \rangle_{d=1}^{ctm[bp,\mathcal{M}_{j_k}]}, s_k]_{k=1}^{m})$$

$\mathcal{U}_i$ computes the witnesses $w_{k,d}$ by running the algorithm $w_{k,d} \leftarrow \mathsf{PProve}(par_p, \phi_{k,d}, c_{k,d}, d_{k,d})$.

$\mathcal{U}_i$ sets the instance as follows.

$$ins \leftarrow (\boxed{par_p}, par_c, pk_1, \mathcal{U}_i, com, bp, [pk_{3,k}, pk_{4,k}, ctm[bp, \mathcal{M}_{j_k}]]_{k=1}^{m}).$$

As for the linear and cumulative policies, the verifying party $\mathcal{P}$, in order to verify the statement $1 = \mathsf{VfSig_3}(pk_{3,k}, s_{k,d}, \langle \mathcal{U}_i, bp, d, c_{k,d}, t_{k,d} \rangle)$ in Line 17 of the relation, must employ values d from 1 to $ctm[bp, \mathcal{M}_{j_k}]$.

4.4 Other policies

In [47], a discrete policy and an interval policy are also considered. In a discrete policy, each consumption value is associated with a price. In an interval policy, each range of consumption values is associated with a price. These policies can be supported by our protocol as simplifications of the linear and cumulative policies. Additionally, it is also possible to consider composite policies created by combining two or more of the aforementioned types.

In the tariff policies considered so far, the price to be paid for a meter reading (c, t) is solely determined by the tariff policy and the values (c, t). However, in many practical tariff policies, the price to be paid depends also on the past behavior of the user. For example, the tariff policies change depending on the last daily or monthly consumption of the user, or on the accumulated consumption of the current day.

Our protocol can support such history-dependent policies as follows. Consider for instance a policy that employs the past consumption pc of the user in the last billing period.

$$\Phi[pc_a, pc_{a+1}](c, t) = \left\{ \begin{array}{ll} \phi_1(c) & \text{if } t \in [t_1, t_2] \\ \vdots & \vdots \\ \phi_L(c) & \text{if } t \in [t_L, t_{L+1}] \end{array} \right\}$$

In this policy, the past consumption pc is divided into intervals $[pc_a, pc_{a+1}]$ for $a \in [1, A]$. Each interval $[pc_a, pc_{a+1}]$ is associated with a spline $\Phi[pc_a, pc_{a+1}](c, t)$, where the price to be paid is determined by a polynomial $\phi_l(c)$ for a meter reading (c, t) such that $t \in [t_l, t_{l+1}]$.

The modification needed in the protocol is as follows. To sign the tariff policy, the service provider signs tuples $[bp, C, t_l, t_{l+1}, pc_a, pc_{a+1}]$, where the the values $[pc_a, pc_{a+1}]$ define a past consumption interval.

In the payment phase, the user computes a commitment com to the past consumption pc of the last billing period and proves in zero-knowledge that pc is correctly computed, i.e., by summing up the consumption values of the meter readings that belong to the last billing period. Then, to compute the proof that the total bill is correct, the user proves knowledge of the value pc in com and proves that $pc \in [pc_a, pc_{a+1}]$ to ensure that the correct commitment C associated with the interval $[pc_a, pc_{a+1}]$ in the tuple $[bp, C, t_l, t_{l+1}, pc_a, pc_{a+1}]$ is employed.

4.5 Efficiency discussion

For a tariff or a cumulative policy, our protocol is quite similar to the protocol provided in [47] for the setting with one meter and one user. In [47], an implementation and performance measurements are provided. Therefore, we refer to [47] for an in-depth efficient analysis.

We analyze now the cost of the protocol proposed in [47] when applying a polynomial tariff policy. To sign the tariff policy, $\mathcal{V}$ computes signatures on tuples $[bp, \phi_0, \phi_1, \ldots, \phi_t, t_l, t_{l+1}, c_m, c_{m+1}]$, where $(\phi_0, \phi_1, \ldots, \phi_t)$ denote the coefficients of the polynomial. In the payment message, the proof of correct evaluation of the polynomial to show that $p = \phi(c)$ employs the coefficients $(\phi_0, \phi_1, \ldots, \phi_t)$. While in our protocol the communication cost of this proof does not depend on the polynomial degree, the cost of this proof grows with the degree.

In Sect. 4.6, we analyze the security of our protocol under two corruption models. Our main analysis considers Byzantine corruptions, where a single adversary corrupts different parties and controls their behavior. Obviously, in this corruption model, when the provider and a meter are corrupt, there is no protocol that can prevent the provider from learning the meter readings input to the meter because both entities are controlled by the same adversary.

For this reason, in Sect. 4.6.7, we also consider a corruption model in which different adversaries, with no communication link between them, corrupt different parties. This model is relevant in the case in which the provider $\mathcal{V}$ and a subset of the meters $\mathcal{M}$ are corrupt, but they cannot communicate directly between each other. In this second corruption model, for the sake of efficiency, the protocol proposed in [47] does not prevent the verifying party from learning the meter readings. The reason is that, in that protocol, instead of proving knowledge of the signatures on the meter readings, the user sends those signatures to the verifying party (the signatures sign commitments to the meter readings, so as not to reveal the meter readings). By manipulating the signature value, a corrupt meter could disclose information on the meter readings to the verifying party.

In [47], it is explained that, to protect user privacy in this corruption model, the user must prove in zero-knowledge possession of the signatures on the meter readings to the verifying party. This is the approach we follow in our protocol, in which the user proves possession of signatures on the meter readings and on the counter of meter readings. Thanks to that,

no information output by the meter to the user is revealed by the user to the verifying party, which allows us to protect user privacy in this corruption model (see Sect. 4.6.7).

4.6 Security analysis of construction BIL

Theorem 1 *Construction* BIL *securely realizes* $\mathcal{F}_{\text{BIL}}$ *in the* $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{REG}}$ *and* $\mathcal{F}_{\text{REG}}^{\text{REG.Ver}}$*-hybrid model.*

We prove that the construction BIL realizes the functionality $\mathcal{F}_{\text{BIL}}$ when a linear, a cumulative and a polynomial policy are employed. We provide a unified description of those proofs. The box POL: … is used to describe a computation that only occurs in the case of a polynomial tariff policy.

To prove that our protocol securely realizes the ideal functionality $\mathcal{F}_{\text{BIL}}$, we have to show that for any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$, such that $\mathcal{Z}$ cannot distinguish whether it is interacting with $\mathcal{A}$ and the protocol in the real world or with $\mathcal{S}$ and $\mathcal{F}_{\text{BIL}}$. The simulator therefore plays the role of all honest parties in the real world and interacts with $\mathcal{F}_{\text{BIL}}$ for all corrupt parties in the ideal world.

Our simulator $\mathcal{S}$ employs any simulator $\mathcal{S}_{\text{CRS}}$, $\mathcal{S}_{\text{SMT}}$, $\mathcal{S}_{\text{REG}}$ and $\mathcal{S}_{\text{REG}}^{\text{REG.Ver}}$ for the constructions that realize the functionalities $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{REG}}$ and $\mathcal{F}_{\text{REG}}^{\text{REG.Ver}}$, respectively. We note that the simulators for all the constructions that realize the functionalities $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{REG}}$ and $\mathcal{F}_{\text{REG}}^{\text{REG.Ver}}$ communicate with each of those functionalities through the same interfaces. These are the interfaces that our simulator employs to communicate with any simulator $\mathcal{S}_{\text{CRS}}$, $\mathcal{S}_{\text{SMT}}$, $\mathcal{S}_{\text{REG}}$ and $\mathcal{S}_{\text{REG}}^{\text{REG.Ver}}$. $\mathcal{S}$ forwards all the messages exchanged between any simulator $\mathcal{S}_{\text{CRS}}$, $\mathcal{S}_{\text{SMT}}$, $\mathcal{S}_{\text{REG}}$ and $\mathcal{S}_{\text{REG}}^{\text{REG.Ver}}$ and the adversary $\mathcal{A}$. When the adversary $\mathcal{A}$ sends a message that corresponds to a protocol that realizes any of the functionalities $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{REG}}$ or $\mathcal{F}_{\text{REG}}^{\text{REG.Ver}}$, $\mathcal{S}$ implicitly forwards that message to the respective simulator $\mathcal{S}_{\text{CRS}}$, $\mathcal{S}_{\text{SMT}}$, $\mathcal{S}_{\text{REG}}$ or $\mathcal{S}_{\text{REG}}^{\text{REG.Ver}}$.

We analyze the case in which the provider $\mathcal{V}$ is corrupt in Sect. 4.6.1. In Sect. 4.6.2, we analyze the case in which a subset of the users $\mathcal{U}$ are corrupt. In Sect. 4.6.3, we analyze the case in which the provider, a subset of the users $\mathcal{U}$ and a subset of the meters $\mathcal{M}$ are corrupt. We provide a detailed analysis of these three cases. Note that the provider or a user can also act as verifying parties, and thus, we consider corrupt verifying parties in all these cases.

We also consider the case in which the provider $\mathcal{V}$ and a subset of the users are corrupt (Sect. 4.6.4), the case in which a subset of the users $\mathcal{U}$ and a subset of the meters $\mathcal{M}$ are corrupt (Sect. 4.6.5) and the case in which the provider $\mathcal{V}$ and a subset of the meters $\mathcal{M}$ are corrupt (Sect. 4.6.6). We do not provide a detailed security analysis of those cases, but describe on a high level the simulator.

We note that, e.g., the case in which only a subset of the users is corrupt is not subsumed by the case in which the provider, a subset of the users and a subset of the meters is corrupt. The reason is that the functionality behaves differently depending on whether the provider is corrupt or not. If the provider and a user are corrupt, the functionality does not guarantee that the price reported by the corrupt user to the verifying party is correct (even if the meters are honest), but when only the user is corrupt, the functionality does guarantee that the price is correct.

When we say that a subset of the users or a subset of the meters is corrupt, we mean that at least one user or at least one meter is corrupt. The security proof does not rely on the fact that the number of corrupt users or the number of corrupt meters is limited by a threshold.

For all the cases above, we consider Byzantine corruptions where a single adversary corrupts different parties and controls their behavior. Obviously, in this corruption model, when the provider and a meter are corrupt, there is no protocol that can prevent the provider from learning the meter readings input to the meter because both entities are controlled by the same adversary.

For this reason, we also consider a corruption model in which different adversaries, with no communication link between them, corrupt different parties. This model is relevant in the case in which the provider $\mathcal{V}$ and a subset of the meters $\mathcal{M}$ are corrupt, but they cannot communicate directly between each other. We show that, under such corruption model, our protocol prevents the corrupt meters from sending information about the meter readings to the verifying parties in Sect. 4.6.7. This is akin to showing that our protocol is collusion-free in the sense of [33].

We note that $\mathcal{F}_{\text{BIL}}$ guarantees that the bill revealed to the verifying party is correct when the user is honest or when the provider and the meters that are involved in the bill computation are honest. For the cases in which a corrupt user colludes with the provider and/or with a meter involved in the computation of the bill, our security analysis shows that our protocol realizes $\mathcal{F}_{\text{BIL}}$, but the total bill revealed to the verifying party is chosen by the adversary.

4.6.1 Case $\mathcal{V}$ corrupt

We start with the case where the provider $\mathcal{V}$ is corrupt. The simulator communicates with the ideal functionality and simulates the behavior of the honest parties toward the corrupt provider. To simulate the behavior of the honest parties, our simulator follows the real-world protocol, with the exception that it creates a simulation trapdoor for the NIPK system and, when an honest user sends a bill to the corrupt provider (which is acting as a verifying party), the simulator computes a simulated non-interactive zero-knowledge proof of knowledge π to create the message (smt.send.end, sid_{SMT}, $\langle p$,

open, *com*, *bp*, $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$, *s*, *ins*, $\pi\rangle$). Therefore, security follows thanks to the zero-knowledge property of the NIPK system. In Fig. 6, we describe our simulator $\mathcal{S}$.

Theorem 2 *When the provider is corrupt, construction* BIL *securely realizes* $\mathcal{F}_{\text{BIL}}$ *in the* $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{REG}}$ *and* $\mathcal{F}_{\text{REG}}^{\text{REG.Ver}}$*-hybrid model if the non-interactive proof of knowledge scheme (*PKSetup, PKProve, PKVerify*) is zero-knowledge.*

Proof We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish between the ensemble $\text{REAL}_{\text{BIL},\mathcal{A},\mathcal{Z}}$ and the ensemble $\text{IDEAL}_{\mathcal{F}_{\text{BIL}},\mathcal{S},\mathcal{Z}}$ with non-negligible probability. We denote by Pr [**Game** *i*] the probability that the environment distinguishes **Game** *i* from the real-world protocol.

Game 0 This game corresponds to the execution of the real-world protocol. Therefore, Pr [**Game** 0] = 0.

Game 1 **Game** 1 follows **Game** 0, except that **Game** 1 computes par_{pk} by running $\mathcal{S}_1(1^k)$. **Game** 1 stores td_s. The zero-knowledge property ensures that par_{pk} output by $\mathcal{S}_1$ is indistinguishable from those output by the algorithm PKSetup. Therefore, $|\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0]| \leq \text{Adv}_{\mathcal{A}}^{\text{zk-nipk}}$.

Game 2 **Game** 2 follows **Game** 1, except that, when an honest user sends a message (smt.send.ini, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$), **Game** 2 computes the proof π by running $\pi \leftarrow \mathcal{S}_2(par_{pk}, td_s, ins)$. The zero-knowledge property ensures that proofs π computed by algorithm $\mathcal{S}_2$ are indistinguishable from those output by PKProve. Therefore, we have that $|\Pr[\textbf{Game } 2] - \Pr[\textbf{Game } 1]| \leq \text{Adv}_{\mathcal{A}}^{\text{zk-nipk}}$.

The distribution of **Game** 2 is identical to that of our simulation.

4.6.2 Case $\mathcal{U}$ corrupt

We analyze the case where a subset of the users $\mathcal{U}_i$ is corrupt. The simulator communicates with the ideal functionality and simulates the behavior of the honest parties toward the subset of corrupt users. To simulate the behavior of the honest parties, our simulator follows the real-world protocol, with two exceptions. First, as in the case where only the provider is corrupt described in Sect. 4.6.1, the simulator creates a simulation trapdoor for the NIPK system and, when an honest user sends a bill to a corrupt user (which is acting as a verifying party), the simulator computes a simulated non-interactive zero-knowledge proof of knowledge π to create the message (smt.send.end, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$). Security follows thanks to the zero-knowledge property of the NIPK system. Second, the simulator aborts when a corrupt user sends a payment message that is verified successfully but where the payment *p* is incorrect. In this case, security follows thanks to the unforgeability of the signature schemes used by the provider, which prevent a dishonest user from forging signatures on the tariff policy or on the list of meters for a billing period, and on the unforgeability of the signature schemes used by the meters, which prevents a dishonest user from forging signatures on meter readings or on the number of readings in a billing period. Additionally, the binding property of the commitment scheme prevents a corrupt user from opening the commitment to the price to an incorrect value. In the case of a polynomial tariff policy, the evaluation binding property of the polynomial commitment scheme prevents a dishonest user from opening the polynomial commitments included in the tariff policy to wrong values. The extraction property of the NIPK scheme is also employed because it is necessary for the simulator to get the signatures and the commitment and polynomial commitment openings included in the witness of the zero-knowledge proof, which is needed to reduce to the unforgeability, binding and evaluation binding properties, respectively. In Figs. 7 and 8, we describe our simulator $\mathcal{S}$.

Theorem 3 *When a subset of the users is corrupt, construction* BIL *securely realizes* $\mathcal{F}_{\text{BIL}}$ *in the* $\mathcal{F}_{\text{CRS}}^{\text{CRS.Setup}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{REG}}$ *and* $\mathcal{F}_{\text{REG}}^{\text{REG.Ver}}$*-hybrid model if the non-interactive proof of knowledge scheme (*PKSetup, PKProve, PKVerify*) is zero-knowledge and extractable, the signature schemes (*$\text{KeyGen}_1, \text{Sign}_1, \text{VfSig}_1$*), (*$\text{KeyGen}_2, \text{Sign}_2, \text{VfSig}_2$*), (*$\text{KeyGen}_3, \text{Sign}_3, \text{VfSig}_3$*), (*$\text{KeyGen}_4, \text{Sign}_4, \text{VfSig}_4$*) are existentially unforgeable, and the commitment scheme (*CSetup, Com, VfCom*) is binding. In the case of a polynomial policy, the polynomial commitment scheme (*PSetup, PCommit, PProve, PVerify*) must be evaluation binding.*

Proof We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish between the ensemble $\text{REAL}_{\text{BIL},\mathcal{A},\mathcal{Z}}$ and the ensemble $\text{IDEAL}_{\mathcal{F}_{\text{BIL}},\mathcal{S},\mathcal{Z}}$ with non-negligible probability. We denote by Pr [**Game** *i*] the probability that the environment distinguishes **Game** *i* from the real-world protocol.

Game 0 This game corresponds to the execution of the real-world protocol. Therefore, Pr [**Game** 0] = 0.

Game 1 **Game** 1 follows **Game** 0, except that, when the adversary sends a message (smt.send.ini, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$) such that *s* is a correct signature on $\langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}\rangle$, but the adversary did not receive any signature s' on $\langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}\rangle$, **Game** 1 aborts. Thanks to the existential unforgeability of

Simulator $\mathcal{S}$: case $\mathcal{V}$ corrupt

The simulator $\mathcal{S}$ employs the simulator $(\mathcal{S}_1, \mathcal{S}_2)$ of the zero-knowledge property of the NIPK scheme described in Section 3.1.

- On input (crs.get.ini, sid) from $\mathcal{S}_{\mathrm{CRS}}$, if (POL: par_p, par_c, par_{pk}, td_s) is not stored, $\mathcal{S}$ runs POL: $par_p \leftarrow \mathsf{PSetup}(1^k, \ell)$, $par_c \leftarrow \mathsf{CSetup}(1^k)$ and $(par_{pk}, td_s) \leftarrow \mathcal{S}_1(1^k)$, and stores (POL: par_p, par_c, par_{pk}, td_s). $\mathcal{S}$ creates a fresh $ssid$, stores $ssid$ and sends (crs.get.sim, sid, $ssid$, ⟨POL: par_p, par_c, par_{pk}⟩) to $\mathcal{S}_{\mathrm{CRS}}$.
- On input the message (crs.get.rep, sid, $ssid$) from $\mathcal{S}_{\mathrm{CRS}}$, if $ssid$ is stored, the simulator $\mathcal{S}$ deletes $ssid$ and sends the message (crs.get.end, sid, ⟨POL: par_p, par_c, par_{pk}⟩) to $\mathcal{S}_{\mathrm{CRS}}$.
- On input the message (reg.register.ini, sid, pk_2) from $\mathcal{S}_{\mathrm{REG}}$, the simulator $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{REG}}$ on input (reg.register.ini, sid, pk_2). When $\mathcal{F}_{\mathrm{REG}}$ outputs the message (reg.register.sim, sid, pk_2), $\mathcal{S}$ sends the message (reg.register.sim, sid, pk_2) to $\mathcal{S}_{\mathrm{REG}}$.
- On input the message (reg.register.rep, sid) from $\mathcal{S}_{\mathrm{REG}}$, the simulator $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{REG}}$ on input the message (reg.register.rep, sid). When $\mathcal{F}_{\mathrm{REG}}$ outputs the message (reg.register.end, sid), the simulator $\mathcal{S}$ sends the message (reg.register.end, sid) to $\mathcal{S}_{\mathrm{REG}}$.
- On input (reg.register.ini, $\langle sid, bp\rangle$, $\langle pk_1, Y_s\rangle$) from $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$ on input (reg.register.ini, $\langle sid, bp\rangle$, $\langle pk_1, Y_s\rangle$). When $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$ outputs (reg.register.sim, $\langle sid, bp\rangle$, $\langle pk_1, Y_s\rangle$), $\mathcal{S}$ retrieves Y from Y_s and sends (bil.policy.ini, sid, bp, Y) to $\mathcal{F}_{\mathrm{BIL}}$. When $\mathcal{F}_{\mathrm{BIL}}$ outputs (bil.policy.sim, sid, bp, Y), $\mathcal{S}$ sends (reg.register.sim, $\langle sid, bp\rangle$, $\langle pk_1, Y_s\rangle$) to $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$.
- On input the message (reg.register.rep, $\langle sid, bp\rangle$) from $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$, the simulator $\mathcal{S}$ runs the copy of $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$ on input the message (reg.register.rep, $\langle sid, bp\rangle$). When $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$ outputs the message (reg.register.end, $\langle sid, bp\rangle$), the simulator $\mathcal{S}$ sends the message (bil.policy.rep, sid, bp) to the functionality $\mathcal{F}_{\mathrm{BIL}}$. When the functionality $\mathcal{F}_{\mathrm{BIL}}$ outputs (bil.policy.end, sid), the simulator $\mathcal{S}$ sends (reg.register.end, $\langle sid, bp\rangle$) to $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$.
- On input the message (smt.send.ini, sid_{SMT}, $\langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s\rangle$) from $\mathcal{S}_{\mathrm{SMT}}$, $\mathcal{S}$ checks that sid_{SMT} is $(\mathcal{V}, \mathcal{U}_i, sid)$. The simulator $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{SMT}}$ on input the message (smt.send.ini, sid_{SMT}, $\langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s\rangle$). When $\mathcal{F}_{\mathrm{SMT}}$ outputs the message (smt.send.sim, sid_{SMT}, $ssid$, $l(\langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s\rangle)$), the simulator $\mathcal{S}$ forwards it to $\mathcal{S}_{\mathrm{SMT}}$.
- On input the message (bil.consumption.sim, sid, $ssid$, $\mathcal{M}_j$, $\mathcal{U}_i$) from the functionality $\mathcal{F}_{\mathrm{BIL}}$, the simulator $\mathcal{S}$ sets $sid_{\mathrm{SMT}} \leftarrow (\mathcal{M}_j, \mathcal{U}_i, sid)$ and sends the message (smt.send.sim, sid_{SMT}, $ssid$, l) to $\mathcal{S}_{\mathrm{SMT}}$, where l is the length of the message $\langle \mathcal{U}_i, bp, d, c, t, s\rangle$.
- On input the message (bil.period.sim, sid, $ssid$, $\mathcal{M}_j$, $\mathcal{U}_i$) from the functionality $\mathcal{F}_{\mathrm{BIL}}$, the simulator $\mathcal{S}$ sets $sid_{\mathrm{SMT}} \leftarrow (\mathcal{M}_j, \mathcal{U}_i, sid)$ and sends the message (smt.send.sim, sid_{SMT}, $ssid$, l) to $\mathcal{S}_{\mathrm{SMT}}$, where l is the length of the message $\langle \mathcal{U}_i, bp, ctm[bp, \mathcal{U}_i], s\rangle$.
- On input (bil.payment.sim, sid, $ssid$, $\mathcal{U}_i$, $\mathcal{P}$) from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ sets $sid_{\mathrm{SMT}} \leftarrow (\mathcal{U}_i, \mathcal{P}, sid)$ and sends (smt.send.sim, sid_{SMT}, $ssid$, l) to $\mathcal{S}_{\mathrm{SMT}}$, where l is the length of the message $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$.
- On input (smt.send.rep, sid_{SMT}, $ssid$) from $\mathcal{S}_{\mathrm{SMT}}$, $\mathcal{S}$ proceeds as follows:
  - If a message (smt.send.sim, sid'_{SMT}, $ssid'$, ...) such that $(sid'_{\mathrm{SMT}}, ssid') = (sid_{\mathrm{SMT}}, ssid)$ was not sent to $\mathcal{S}_{\mathrm{SMT}}$, $\mathcal{S}$ ignores the message.
  - Else, if the message (smt.send.sim, sid_{SMT}, $ssid$, ...) was sent after receiving a message (smt.send.ini, sid_{SMT}, $\langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s\rangle$) from $\mathcal{S}_{\mathrm{SMT}}$, $\mathcal{S}$ runs the corresponding instance of $\mathcal{F}_{\mathrm{SMT}}$ on input (smt.send.rep, sid_{SMT}, $ssid$). When $\mathcal{F}_{\mathrm{SMT}}$ sends (smt.send.end, sid_{SMT}, $\langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s\rangle$), $\mathcal{S}$ does nothing if there is not an instance of $\mathcal{F}_{\mathrm{REG}}$ that stores pk_2. $\mathcal{S}$ does nothing if a tuple $(sid, bp', \mathcal{U}'_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s)$ such that $bp' = bp$ and $\mathcal{U}'_i = \mathcal{U}_i$ is already stored. $\mathcal{S}$ does nothing if $1 \neq \mathsf{VfSig}_2(pk_2, s, \langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}\rangle)$. Otherwise $\mathcal{S}$ stores $(sid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s)$. $\mathcal{S}$ sends (bil.listmeters.ini, sid, bp, $\mathcal{U}_i$, $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$) to $\mathcal{F}_{\mathrm{BIL}}$. When $\mathcal{F}_{\mathrm{BIL}}$ outputs (bil.listmeters.sim, sid, $ssid$, $\mathcal{U}_i$), $\mathcal{S}$ sends (bil.listmeters.rep, sid, $ssid$) to $\mathcal{F}_{\mathrm{BIL}}$.
  - Else, if the message (smt.send.sim, sid_{SMT}, $ssid$, ...) was sent after receiving a message (bil.consumption.sim, sid, $ssid$, $\mathcal{M}_j$, $\mathcal{U}_i$) from the functionality $\mathcal{F}_{\mathrm{BIL}}$, the simulator $\mathcal{S}$ sends the message (bil.consumption.rep, sid, $ssid$) to the functionality $\mathcal{F}_{\mathrm{BIL}}$.
  - Else, if the message (smt.send.sim, sid_{SMT}, $ssid$, ...) was sent after receiving a message (bil.period.sim, sid, $ssid$, $\mathcal{M}_j$, $\mathcal{U}_i$) from the functionality $\mathcal{F}_{\mathrm{BIL}}$, the simulator $\mathcal{S}$ sends the message (bil.period.rep, sid, $ssid$) to the functionality $\mathcal{F}_{\mathrm{BIL}}$.
  - Else, if the message (smt.send.sim, sid_{SMT}, $ssid$, ...) was sent after receiving a message (bil.payment.sim, sid, $ssid$, $\mathcal{U}_i$, $\mathcal{P}$) from $\mathcal{F}_{\mathrm{BIL}}$, the simulator $\mathcal{S}$ sends the message (bil.payment.rep, sid, $ssid$) to the functionality $\mathcal{F}_{\mathrm{BIL}}$.
- On input (bil.payment.end, sid, $\mathcal{U}_i$, bp, $p[bp]$, $\mathcal{M}_{j_1}, N[\mathcal{M}_{j_1}, bp], \ldots, \mathcal{M}_{j_m}, N[\mathcal{M}_{j_m}, bp]$) from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ proceeds as follows:
  - $\mathcal{S}$ retrieves the stored POL: par_p, par_c and (par_{pk}, td_s).
  - $\mathcal{S}$ retrieves pk_1 and Y_s for the billing period bp from the corresponding copy of $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$.
  - If $(sk_{3,k}, pk_{3,k})$ and $(sk_{4,k}, pk_{4,k})$ are not stored, for $k = 1$ to m, $\mathcal{S}$ runs $(sk_{3,k}, pk_{3,k}) \leftarrow \mathsf{KeyGen}_3(1^k)$ and $(sk_{4,k}, pk_{4,k}) \leftarrow \mathsf{KeyGen}_4(1^k)$ and stores $(sk_{3,k}, pk_{3,k})$ and $(sk_{4,k}, pk_{4,k})$.
  - $\mathcal{S}$ runs $(com, open) \leftarrow \mathsf{Com}(par_c, p)$.
  - $\mathcal{S}$ sets $ins \leftarrow$ (POL: par_p, par_c, pk_1, $\mathcal{U}_i$, com, bp, $[pk_{3,k}, pk_{4,k}, ctm[bp, \mathcal{M}_{j_k}]]_{k=1}^m$).
  - $\mathcal{S}$ runs $\pi \leftarrow \mathcal{S}_2(par_{pk}, td_s, ins)$. The relation R used by $\mathcal{S}_2$ is described in Section 4.1, Section 4.2 and Section 4.3.
  - $\mathcal{S}$ recovers sid_{SMT} from the last (smt.send.rep, sid_{SMT}, $ssid$) message received from $\mathcal{S}_{\mathrm{SMT}}$. $\mathcal{S}$ sends (smt.send.end, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$) to $\mathcal{S}_{\mathrm{SMT}}$.

Fig. 6 Simulator $\mathcal{S}$: case $\mathcal{V}$ corrupt

Simulator $\mathcal{S}$: case $\mathcal{U}$ corrupt (I)

The simulator $\mathcal{S}$ employs the simulator $(\mathcal{S}_1, \mathcal{S}_2)$ of the zero-knowledge property of the NIPK scheme described in Section 3.1.

- On input $(\mathsf{bil.policy.sim}, sid, bp, Y)$ from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ proceeds as follows:
  - If this is the first $(\mathsf{bil.policy.sim}, sid, \ldots)$ message received from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ does the following. If $(\boxed{\text{POL: } par_p}, par_c, par_{pk}, td_s)$ are not stored, $\mathcal{S}$ runs the algorithms $\boxed{\text{POL: } par_p \leftarrow \mathsf{PSetup}(1^k, \ell),}$ $par_c \leftarrow \mathsf{CSetup}(1^k)$ and $(par_{pk}, td_s) \leftarrow \mathcal{S}_1(1^k)$, and stores $(\boxed{\text{POL: } par_p}, par_c, par_{pk}, td_s)$. $\mathcal{S}$ creates a fresh $ssid$, stores $(ssid, bp, Y)$, and sends $(\mathsf{crs.get.sim}, sid, ssid, \langle \boxed{\text{POL: } par_p}, par_c, par_{pk} \rangle)$ to $\mathcal{S}_{\mathrm{CRS}}$.
  - Else, $\mathcal{S}$ computes a signed tariff policy Y_s for Y as described in Section 4.1, Section 4.2 or Section 4.3. $\mathcal{S}$ stores (bp, Y_s) and sends $(\mathsf{reg.register.sim}, \langle sid, bp \rangle, \langle pk_1, Y_s \rangle)$ to $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$.
- On input $(\mathsf{crs.get.ini}, sid)$ from $\mathcal{S}_{\mathrm{CRS}}$, $\mathcal{S}$ works as in the case where $\mathcal{V}$ is corrupt.
- On input the message $(\mathsf{crs.get.rep}, sid, ssid)$ from $\mathcal{S}_{\mathrm{CRS}}$, $\mathcal{S}$ proceeds as follows:
  - If there is a tuple $(ssid', bp, Y)$ such that $ssid = ssid'$, $\mathcal{S}$ proceeds as follows. $\mathcal{S}$ runs $(sk_1, pk_1) \leftarrow \mathsf{KeyGen}_1(1^k)$ and stores (sk_1, pk_1). $\mathcal{S}$ computes a signed tariff policy Y_s for Y as described in Section 4.1, Section 4.2 or Section 4.3. $\mathcal{S}$ stores (bp, Y_s), deletes $(ssid, bp, Y)$ and sends $(\mathsf{reg.register.sim}, \langle sid, bp \rangle, \langle pk_1, Y_s \rangle)$ to $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$.
  - Else, $\mathcal{S}$ sends the message $(\mathsf{crs.get.end}, sid, \langle \boxed{\text{POL: } par_p}, par_c, par_{pk} \rangle)$ to $\mathcal{S}_{\mathrm{CRS}}$.
- On input $(\mathsf{reg.register.rep}, \langle sid, bp \rangle)$ from $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$, if (bp, Y_s) is stored, $\mathcal{S}$ sends $(\mathsf{bil.policy.rep}, sid, bp)$ to $\mathcal{F}_{\mathrm{BIL}}$.
- On input $(\mathsf{reg.retrieve.ini}, \langle sid, bp \rangle)$ from $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$, $\mathcal{S}$ creates a fresh $ssid$. If $\langle pk_1, Y_s \rangle$ are not stored, $\mathcal{S}$ stores $(sid, bp, ssid, \perp)$ and sends $(\mathsf{reg.retrieve.sim}, \langle sid, bp \rangle, ssid, \perp)$ to $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$, else stores $(sid, bp, ssid, \langle pk_1, Y_s \rangle)$ and sends $(\mathsf{reg.retrieve.sim}, \langle sid, bp \rangle, ssid, \langle pk_1, Y_s \rangle)$ to $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$.
- On input $(\mathsf{reg.retrieve.rep}, \langle sid, bp \rangle, ssid)$ from $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$, $\mathcal{S}$ ignores the message if there is no tuple $(sid, bp, ssid, \ldots)$ stored. If there is a tuple $(sid, bp, ssid, \perp)$ stored, $\mathcal{S}$ sends $(\mathsf{reg.retrieve.end}, \langle sid, bp \rangle, \perp)$ to $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$. If there is a tuple $(sid, bp, ssid, \langle pk_1, Y_s \rangle)$ stored, $\mathcal{S}$ sends $(\mathsf{reg.retrieve.end}, \langle sid, bp \rangle, \langle pk_1, Y_s \rangle)$ to $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$.
- On input $(\mathsf{bil.listmeters.sim}, sid, ssid, \mathcal{U}_i)$ from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ sets $sid_{\mathrm{SMT}} \leftarrow (\mathcal{V}, \mathcal{U}_i, sid)$ and sends $(\mathsf{smt.send.sim}, sid_{\mathrm{SMT}}, ssid, l)$ to $\mathcal{S}_{\mathrm{SMT}}$, where l is the length of the message $\langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s \rangle$.
- On input $(\mathsf{bil.listmeters.end}, sid, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m})$ from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ proceeds as follows. If (sk_2, pk_2) is not stored, $\mathcal{S}$ runs $(sk_2, pk_2) \leftarrow \mathsf{KeyGen}_2(1^k)$ and stores (sk_2, pk_2). $\mathcal{S}$ signs $s \leftarrow \mathsf{Sign}_2(sk_2, \langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m} \rangle)$. $\mathcal{S}$ uses the last sid_{SMT} received in a $(\mathsf{smt.send.rep}, sid_{\mathrm{SMT}}, ssid)$ message from $\mathcal{S}_{\mathrm{SMT}}$ and sends $(\mathsf{smt.send.end}, sid_{\mathrm{SMT}}, \langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s \rangle)$ to $\mathcal{S}_{\mathrm{SMT}}$.
- On input $(\mathsf{reg.retrieve.ini}, sid)$ from $\mathcal{S}_{\mathrm{REG}}$, $\mathcal{S}$ creates a fresh $ssid$. If (sk_2, pk_2) is not stored, $\mathcal{S}$ stores $(ssid, \perp)$ and sends $(\mathsf{reg.retrieve.sim}, sid, ssid, \perp)$ to $\mathcal{S}_{\mathrm{REG}}$, else stores $(ssid, pk_2)$ and sends $(\mathsf{reg.retrieve.sim}, sid, ssid, pk_2)$ to $\mathcal{S}_{\mathrm{REG}}$.
- On input $(\mathsf{reg.retrieve.rep}, sid, ssid)$ from $\mathcal{S}_{\mathrm{REG}}$, $\mathcal{S}$ ignores the message if there is no tuple $(ssid, \ldots)$ stored. If there is a tuple $(ssid, \perp)$ stored, $\mathcal{S}$ sends $(\mathsf{reg.retrieve.end}, sid, \perp)$ to $\mathcal{S}_{\mathrm{REG}}$. If there is a tuple $(ssid, pk_2)$ stored, $\mathcal{S}$ sends $(\mathsf{reg.retrieve.end}, sid, pk_2)$ to $\mathcal{S}_{\mathrm{REG}}$.
- On input $(\mathsf{bil.consumption.sim}, sid, ssid, \mathcal{M}_j, \mathcal{U}_i)$ from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ sets $sid_{\mathrm{SMT}} \leftarrow (\mathcal{M}_j, \mathcal{U}_i, sid)$ and sends $(\mathsf{smt.send.sim}, sid_{\mathrm{SMT}}, ssid, l)$ to $\mathcal{S}_{\mathrm{SMT}}$, where l is the length of the message $\langle \mathcal{U}_i, bp, d, c, t, s \rangle$.
- On input $(\mathsf{bil.consumption.end}, sid, \mathcal{M}_j, bp, c, t)$ from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ proceeds as follows. If $(sk_{3,k}, pk_{3,k})$ and $(sk_{4,k}, pk_{4,k})$ are not stored for the meter $\mathcal{M}_j$, $\mathcal{M}_j$ runs $(sk_{3,k}, pk_{3,k}) \leftarrow \mathsf{KeyGen}_3(1^k)$ and $(sk_{4,k}, pk_{4,k}) \leftarrow \mathsf{KeyGen}_4(1^k)$, and stores $(sk_{3,k}, pk_{3,k})$ and $(sk_{4,k}, pk_{4,k})$. $\mathcal{S}$ recovers sid_{SMT} from the last $(\mathsf{smt.send.rep}, sid_{\mathrm{SMT}}, ssid)$ message sent by $\mathcal{S}_{\mathrm{SMT}}$. $\mathcal{S}$ gets $\mathcal{U}_i$ from sid_{SMT}. $\mathcal{S}$ increments a counter $ctm[bp, \mathcal{M}_j, \mathcal{U}_i]$ (initialized at zero) that counts the number of meter readings that $\mathcal{M}_j$ sends to $\mathcal{U}_i$ during the billing period bp. $\mathcal{S}$ runs $s \leftarrow \mathsf{Sign}_3(sk_{3,k}, \langle \mathcal{U}_i, bp, ctm[bp, \mathcal{M}_j, \mathcal{U}_i], c, t \rangle)$. $\mathcal{S}$ sends $(\mathsf{smt.send.end}, sid_{\mathrm{SMT}}, \langle \mathcal{U}_i, bp, ctm[bp, \mathcal{M}_j, \mathcal{U}_i], c, t, s \rangle)$ to $\mathcal{S}_{\mathrm{SMT}}$.
- On input $(\mathsf{reg.retrieve.ini}, \langle sid, \mathcal{M}_j \rangle)$ from $\mathcal{S}_{\mathrm{REG}}$, $\mathcal{S}$ creates a fresh $ssid$. If $(sk_{3,k}, pk_{3,k})$ and $(sk_{4,k}, pk_{4,k})$ are not stored, $\mathcal{S}$ stores $(ssid, \perp)$ and sends $(\mathsf{reg.retrieve.sim}, \langle sid, \mathcal{M}_j \rangle, ssid, \perp)$ to $\mathcal{S}_{\mathrm{REG}}$, else stores $(ssid, \langle pk_{3,k}, pk_{4,k} \rangle)$ and sends $(\mathsf{reg.retrieve.sim}, \langle sid, \mathcal{M}_j \rangle, ssid, \langle pk_{3,k}, pk_{4,k} \rangle)$ to $\mathcal{S}_{\mathrm{REG}}$.
- On input $(\mathsf{reg.retrieve.rep}, \langle sid, \mathcal{M}_j \rangle, ssid)$ from $\mathcal{S}_{\mathrm{REG}}$, $\mathcal{S}$ ignores the message if there is no tuple $(ssid, \ldots)$ stored. If there is a tuple $(ssid, \perp)$ stored, $\mathcal{S}$ sends $(\mathsf{reg.retrieve.end}, \langle sid, \mathcal{M}_j \rangle, \perp)$ to $\mathcal{S}_{\mathrm{REG}}$. If there is a tuple $(ssid, \langle pk_{3,k}, pk_{4,k} \rangle)$ stored, $\mathcal{S}$ sends $(\mathsf{reg.retrieve.end}, \langle sid, \mathcal{M}_j \rangle, \langle pk_{3,k}, pk_{4,k} \rangle)$ to $\mathcal{S}_{\mathrm{REG}}$.
- On input $(\mathsf{bil.period.sim}, sid, ssid, \mathcal{M}_j, \mathcal{U}_i)$ from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ sets $sid_{\mathrm{SMT}} \leftarrow (\mathcal{M}_j, \mathcal{U}_i, sid)$ and sends $(\mathsf{smt.send.sim}, sid_{\mathrm{SMT}}, ssid, l)$ to $\mathcal{S}_{\mathrm{SMT}}$, where l is the length of the message $\langle \mathcal{U}_i, bp, ctm[bp, \mathcal{M}_j], s \rangle$.
- On input $(\mathsf{bil.period.end}, sid, bp, \mathcal{M}_j, N[\mathcal{M}_j, bp])$ from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ proceeds as follows. If $(sk_{3,k}, pk_{3,k})$ and $(sk_{4,k}, pk_{4,k})$ are not stored for $\mathcal{M}_j$, $\mathcal{S}$ runs $(sk_{3,k}, pk_{3,k}) \leftarrow \mathsf{KeyGen}_3(1^k)$ and $(sk_{4,k}, pk_{4,k}) \leftarrow \mathsf{KeyGen}_4(1^k)$, and stores $(sk_{3,k}, pk_{3,k})$ and $(sk_{4,k}, pk_{4,k})$. $\mathcal{S}$ recovers sid_{SMT} from the last $(\mathsf{smt.send.rep}, sid_{\mathrm{SMT}}, ssid)$ message sent by $\mathcal{S}_{\mathrm{SMT}}$. $\mathcal{S}$ gets sid_{SMT} and $\mathcal{U}_i$ from sid_{SMT}. $\mathcal{S}$ runs $s \leftarrow \mathsf{Sign}_4(sk_{4,k}, \langle \mathcal{U}_i, bp, N[\mathcal{M}_j, bp] \rangle)$. $\mathcal{S}$ sends $(\mathsf{smt.send.end}, sid_{\mathrm{SMT}}, \langle \mathcal{U}_i, bp, N[\mathcal{M}_j, bp], s \rangle)$ to $\mathcal{S}_{\mathrm{SMT}}$.
- On input $(\mathsf{bil.payment.sim}, sid, ssid, \mathcal{U}_i, \mathcal{P})$ from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ sets $sid_{\mathrm{SMT}} \leftarrow (\mathcal{U}_i, \mathcal{P}, sid)$ and sends $(\mathsf{smt.send.sim}, sid_{\mathrm{SMT}}, ssid, l)$ to $\mathcal{S}_{\mathrm{SMT}}$, where l is the length of the message $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi \rangle$.
- On input $(\mathsf{bil.payment.end}, sid, \mathcal{U}_i, bp, p[bp], \mathcal{M}_{j_1}, N[\mathcal{M}_{j_1}, bp], \ldots, \mathcal{M}_{j_m}, N[\mathcal{M}_{j_m}, bp])$ from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ proceeds as in the case where $\mathcal{V}$ is corrupt, except that $\mathcal{S}$ replaces $\mathcal{V}$ by the identity of the corrupt user that acts as verifying party.

Fig. 7 Simulator $\mathcal{S}$: case $\mathcal{U}$ corrupt (I)

Simulator $\mathcal{S}$: case $\mathcal{U}$ corrupt (II)

- On input ($\mathsf{smt.send.ini}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$) from $\mathcal{S}_{\mathrm{SMT}}$, $\mathcal{S}$ proceeds as follows.
  - $\mathcal{S}$ ignores the message if (sk_2, pk_2) or (sk_2, pk_2) are not stored, or if $(sk_{3,k}, pk_{3,k})$ or $(sk_{4,k}, pk_{4,k})$ are not stored for any meter $\mathcal{M}_{j_k}$ (for $k = 1$ to m), or if $\boxed{\text{POL: } par_p,}$ par_c and par_{pk} are not stored.
  - $\mathcal{S}$ parses sid_{SMT} as $(\mathcal{U}_i, \mathcal{P}, sid)$.
  - $\mathcal{S}$ checks if the values $\boxed{\text{POL: } par_p,}$ par_c, pk_1, $pk_{3,k}$, and $pk_{4,k}$ stored are equal to those in the instance $ins = (\boxed{\text{POL: } par_p,} par_c, pk_1, \mathcal{U}_i, com, bp, [pk_{3,k}, pk_{4,k}, ctm[bp, \mathcal{M}_{j_k}]]_{k=1}^{m})$. If not, $\mathcal{S}$ ignores the message. $\mathcal{S}$ also checks that, for $k = 1$ to m, the instance includes a counter $ctm[bp, \mathcal{M}_{j_k}]$ of meter readings and that the proof proves possession of $ctm[bp, \mathcal{M}_{j_k}]$ meter readings numbered from 1 to $ctm[bp, \mathcal{M}_{j_k}]$.
  - $\mathcal{S}$ ignores the message if $1 \neq \mathsf{VfCom}(par_c, com, p, open)$.
  - $\mathcal{S}$ ignores the message if $1 \neq \mathsf{VfSig}_2(pk_2, s, \langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}\rangle)$.
  - $\mathcal{S}$ aborts if the adversary did not receive any signature s on $\langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}\rangle$.
  - $\mathcal{S}$ ignores the message if $1 \neq \mathsf{PKVerify}(par_{pk}, ins, \pi)$.
  - $\mathcal{S}$ aborts if, for $k = 1$ to m, the adversary did not receive a signature s_k on the tuple $\langle bp, \mathcal{U}_i, ctm[bp, \mathcal{M}_{j_k}]\rangle$.
  - $\mathcal{S}$ performs the computation of the price p' to be paid by $\mathcal{U}_i$ at the billing period bp using as input the meter readings sent to $\mathcal{U}_i$ by the meters $\langle \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}\rangle$ and the tariff policy for that billing period. If $p' \neq p$, $\mathcal{S}$ aborts.
  - $\mathcal{S}$ gets the verifying party identifier $\mathcal{P}$ from sid_{SMT} and sends ($\mathsf{bil.payment.ini}$, sid, $\mathcal{P}$, bp) to $\mathcal{F}_{\mathrm{BIL}}$.
- On input ($\mathsf{smt.send.rep}$, sid_{SMT}, $ssid$) from $\mathcal{S}_{\mathrm{SMT}}$, $\mathcal{S}$ proceeds as follows.
  - If a message ($\mathsf{smt.send.sim}$, sid'_{SMT}, $ssid'$, ...) such that $(sid_{\mathrm{SMT}}, ssid) = (sid'_{\mathrm{SMT}}, ssid')$ was not sent to $\mathcal{S}_{\mathrm{SMT}}$, $\mathcal{S}$ ignores the message.
  - Else, if the message ($\mathsf{smt.send.sim}$, sid_{SMT}, $ssid$, ...) was sent after receiving a message ($\mathsf{bil.listmeters.sim}$, sid, $ssid$, $\mathcal{U}_i$) from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ sends ($\mathsf{bil.listmeters.rep}$, sid, $ssid$) to $\mathcal{F}_{\mathrm{BIL}}$.
  - Else, if the message ($\mathsf{smt.send.sim}$, sid_{SMT}, $ssid$, ...) was sent after receiving a message ($\mathsf{bil.consumption.sim}$, sid, $ssid$, $\mathcal{M}_j$, $\mathcal{U}_i$) from the functionality $\mathcal{F}_{\mathrm{BIL}}$, the simulator $\mathcal{S}$ sends the message ($\mathsf{bil.consumption.rep}$, sid, $ssid$) to the functionality $\mathcal{F}_{\mathrm{BIL}}$.
  - Else, if the message ($\mathsf{smt.send.sim}$, sid_{SMT}, $ssid$, ...) was sent after receiving a message ($\mathsf{bil.period.sim}$, sid, $ssid$, $\mathcal{M}_j$, $\mathcal{U}_i$) from the functionality $\mathcal{F}_{\mathrm{BIL}}$, the simulator $\mathcal{S}$ sends the message ($\mathsf{bil.period.rep}$, sid, $ssid$) to the functionality $\mathcal{F}_{\mathrm{BIL}}$.
  - Else, if the message ($\mathsf{smt.send.sim}$, sid_{SMT}, $ssid$, ...) was sent after receiving a message ($\mathsf{bil.payment.sim}$, sid, $ssid$, $\mathcal{U}_i$, $\mathcal{P}$) from $\mathcal{F}_{\mathrm{BIL}}$, the simulator $\mathcal{S}$ sends the message ($\mathsf{bil.payment.rep}$, sid, $ssid$) to the functionality $\mathcal{F}_{\mathrm{BIL}}$.

Fig. 8 Simulator $\mathcal{S}$: case $\mathcal{U}$ corrupt (II)

the signature scheme (KeyGen_2, Sign_2, VfSig_2), **Game** 1 aborts with negligible probability. Therefore, $|\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{unf-sig}}$.

Game 2 **Game** 2 follows **Game** 1, except that **Game** 2 computes the parameters par_{pk} by running $(par_{pk}, td_e) \leftarrow \mathcal{E}_1(1^k)$. **Game** 2 stores td_e. The extraction property ensures that the parameters par_{pk} output by $\mathcal{E}_1(1^k)$ is indistinguishable from those output by $\mathsf{PKSetup}$. Therefore, $|\Pr[\textbf{Game } 2] - \Pr[\textbf{Game } 1]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{ex-nipk}}$.

Game 3 **Game** 3 follows **Game** 2, except that, when the adversary sends ($\mathsf{smt.send.ini}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$), after verifying s, com and π, **Game** 3 runs $wit \leftarrow \mathcal{E}_2(par_{pk}, td_e, ins, \pi)$. **Game** 3 aborts if extraction fails. The extraction property ensures that extraction works with overwhelming probability. Therefore, $|\Pr[\textbf{Game } 3] - \Pr[\textbf{Game } 2]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{ex-nipk}}$.

Game 4 **Game** 4 follows **Game** 3, except that, when the adversary sends ($\mathsf{smt.send.ini}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$), after extracting the witness wit, **Game** 4 aborts if any of the signatures s_k in the witness wit signs a tuple $\langle bp, \mathcal{U}_i, ctm[bp, \mathcal{M}_{j_k}]\rangle$ such that no signature on that tuple was sent to the adversary. Thanks to the existential unforgeability of the signature scheme (KeyGen_4, Sign_4, VfSig_4), **Game** 4 aborts with negligible probability. Therefore, $|\Pr[\textbf{Game } 4] - \Pr[\textbf{Game } 3]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{unf-sig}}$.

Game 5 **Game** 5 follows **Game** 4, except that, when the adversary sends a message ($\mathsf{smt.send.ini}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$), after extracting the witness wit, $\mathcal{S}$ aborts if any of the signatures $s_{k,d}$ in the witness wit signs a tuple $\langle \mathcal{U}_i, bp, d, c_{k,d}, t_{k,d}\rangle$ such that a signature on that tuple was not sent to the adversary. Thanks to the existential unforgeability of the signature scheme (KeyGen_3, Sign_3, VfSig_3), **Game** 5 aborts with negligible probability. Therefore, $|\Pr[\textbf{Game } 5] - \Pr[\textbf{Game } 4]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{unf-sig}}$.

Game 6 **Game** 6 follows **Game** 5, except that, when the adversary sends a message ($\mathsf{smt.send.ini}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$), after extracting the witness wit, $\mathcal{S}$ aborts if any of the signatures $s'_{k,d}$ in the witness wit signs a tuple such that the tuple is not in the signed policy Y_s sent to

the adversary. The tuple is of one of the following forms.

Linear Policy The tuple is of the form $\langle bp, r_{k,d}, t_{min,k,d}, t_{max,k,d}\rangle$.
Cumulative Policy The tuple is of the form $\langle bp, r_{k,d}, F_{k,d}, t_{min,k,d}, t_{max,k,d}, c_{min,k,d}, c_{max,k,d}\rangle$.
Polynomial Policy. The tuple is of the form $\langle bp, C_{k,d}, t_{min,k,d}, t_{max,k,d}, c_{min,k,d}, c_{max,k,d}\rangle$.

Thanks to the existential unforgeability of the signature scheme ($\mathsf{KeyGen}_1, \mathsf{Sign}_1, \mathsf{VfSig}_1$), **Game** 6 aborts with negligible probability. Therefore, we have that $|\Pr[\textbf{Game } 6] - \Pr[\textbf{Game } 5]| \leq \mathrm{Adv}^{\mathsf{unf-sig}}_{\mathcal{A}}$.

Polynomial policy only:

Game 7 **Game** 7 follows **Game** 6, except that, when the adversary sends a message ($\mathsf{smt.send.ini}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$), after extracting the witness wit, **Game** 7 aborts if any price $p_{k,d}$ in the witness wit is not the result of evaluating on input the consumption $c_{k,d}$ the polynomial in the policy Y_s associated with the time interval $[t_{min,k,d}, t_{max,k,d})$ and the consumption interval $[c_{min,k,d}, c_{max,k,d})$ such that $t_{k,d} \in [t_{min,k,d}, t_{max,k,d})$ and $c_{k,d} \in [c_{min,k,d}, c_{max,k,d})$. The evaluation binding property of the polynomial commitment scheme prevents a polynomial commitment from being open on the same input to two different values. Therefore, $|\Pr[\textbf{Game } 7] - \Pr[\textbf{Game } 6]| \leq \mathrm{Adv}^{\mathsf{bd-pcom}}_{\mathcal{A}}$.

Game 8 **Game** 8 follows **Game** 7, except that, when the adversary sends a message ($\mathsf{smt.send.ini}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$), after extracting the witness wit, $\mathcal{S}$ aborts if $(p', open')$ in the witness wit does not equal $(p, open)$. The binding property of the commitment scheme prevents the commitment com from being opened to two different values. Therefore, we have that $|\Pr[\textbf{Game } 8] - \Pr[\textbf{Game } 7]| \leq \mathrm{Adv}^{\mathsf{bd-com}}_{\mathcal{A}}$.

In **Game** 8, we have shown that $\mathcal{S}$ receives with negligible probability a message ($\mathsf{smt.send.ini}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$) that makes $\mathcal{S}$ abort.

Game 9 **Game** 9 follows **Game** 8, except that **Game** 9 computes par_{pk} by running $\mathcal{S}_1(1^k)$. **Game** 9 stores td_s. **Game** 9 does not run the extractor $\mathcal{E}_2$. The zero-knowledge property ensures that par_{pk} output by $\mathcal{S}_1$ is indistinguishable from those output by $\mathsf{PKSetup}$. Therefore, $|\Pr[\textbf{Game } 9] - \Pr[\textbf{Game } 8]| \leq \mathrm{Adv}^{\mathsf{zk-nipk}}_{\mathcal{A}}$.

Game 10 **Game** 10 follows **Game** 9, except that, when an honest user sends ($\mathsf{smt.send.ini}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$), **Game** 10 computes the proof π by running $\pi \leftarrow \mathcal{S}_2(par_{pk}, td_s, ins)$. The zero-knowledge property ensures that proofs π computed by algorithm $\mathcal{S}_2$ are indistinguishable from those output by $\mathsf{PKProve}$. Therefore, $|\Pr[\textbf{Game } 10] - \Pr[\textbf{Game } 9]| \leq \mathrm{Adv}^{\mathsf{zk-nipk}}_{\mathcal{A}}$.

The distribution of **Game** 10 is identical to that of our simulation.

4.6.3 Case $\mathcal{V}, \mathcal{U}$ and $\mathcal{M}$ corrupt

We analyze the case where the provider $\mathcal{V}$, a subset of the users $\mathcal{U}_i$ and a subset of the meters $\mathcal{M}_j$ are corrupt. The simulator communicates with the ideal functionality and simulates the behavior of the honest parties toward the subset of corrupt users, the subset of corrupt meters and the corrupt provider. To simulate the behavior of the honest parties, our simulator follows the real-world protocol, with two exceptions. First, as in the cases where only the provider is corrupt described in Sect. 4.6.1 and where a subset of the users is corrupt in Sect. 4.6.2, the simulator creates a simulation trapdoor for the NIPK system and, when an honest user sends a bill to a corrupt verifying party, the simulator computes a simulated non-interactive zero-knowledge proof of knowledge π to create the message ($\mathsf{smt.send.end}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$). Security follows thanks to the zero-knowledge property of the NIPK system. Second, the simulator aborts when a corrupt user sends a payment message that is verified successfully, but where, for any of the meters in the payment message, if the meter is honest, the number of meter readings included in the instance ins is not correct. In this case, security follows thanks to the unforgeability of the signature scheme that meters use to sign the number of meter readings. The extraction property of the NIPK scheme is also employed because it is necessary for the simulator to get the signatures on the number of meter readings included in the witness of the zero-knowledge proof, which is needed to reduce to the unforgeability property. In Figs. 9 and 10, we describe our simulator $\mathcal{S}$.

Theorem 4 *When the provider $\mathcal{V}$, a subset of the users and a subset of the meters are corrupt, construction* BIL *securely realizes $\mathcal{F}_{\mathrm{BIL}}$ in the $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}, \mathcal{F}_{\mathrm{SMT}}, \mathcal{F}_{\mathrm{REG}}$ and $\mathcal{F}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$-hybrid model if the non-interactive proof of knowl-*

Simulator $\mathcal{S}$: case $\mathcal{V}, \mathcal{U}$ and $\mathcal{M}$ corrupt (I)

- On input (crs.get.ini, sid) from $\mathcal{S}_{\mathrm{CRS}}$, $\mathcal{S}$ works as in the case where $\mathcal{V}$ is corrupt.
- On input (crs.get.rep, sid, $ssid$) from $\mathcal{S}_{\mathrm{CRS}}$, $\mathcal{S}$ works as in the case where $\mathcal{V}$ is corrupt.
- On input (reg.register.ini, sid, pk_2) from $\mathcal{S}_{\mathrm{REG}}$, $\mathcal{S}$ works as in the case where $\mathcal{V}$ is corrupt.
- On input (reg.register.rep, sid) from $\mathcal{S}_{\mathrm{REG}}$, $\mathcal{S}$ works as in the case where $\mathcal{V}$ is corrupt.
- On input (reg.register.ini, $\langle sid, bp\rangle$, $\langle pk_1, Y_s\rangle$) from $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$, $\mathcal{S}$ works as in the case where $\mathcal{V}$ is corrupt.
- On input (reg.register.rep, $\langle sid, bp\rangle$) from $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$, $\mathcal{S}$ works as in the case where $\mathcal{V}$ is corrupt.
- On input (reg.retrieve.ini, sid) from $\mathcal{S}_{\mathrm{REG}}$, $\mathcal{S}$ works as in the case where $\mathcal{U}$ is corrupt.
- On input (reg.retrieve.rep, sid, $ssid$) from $\mathcal{S}_{\mathrm{REG}}$, $\mathcal{S}$ works as in the case where $\mathcal{U}$ is corrupt.
- On input (reg.retrieve.ini, $\langle sid, bp\rangle$) from $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$, $\mathcal{S}$ works as in the case where $\mathcal{U}$ is corrupt.
- On input (reg.retrieve.rep, $\langle sid, bp\rangle$, $ssid$) from $\mathcal{S}_{\mathrm{REG}}^{\mathsf{REG.Ver}}$, $\mathcal{S}$ works as in the case where $\mathcal{U}$ is corrupt.
- On input (smt.send.ini, sid_{SMT}, $\langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s\rangle$) from $\mathcal{S}_{\mathrm{SMT}}$, $\mathcal{S}$ works as in the case where $\mathcal{V}$ is corrupt.
- On input (reg.register.ini, $\langle sid, \mathcal{M}_j\rangle$, $\langle pk_{3,k}, pk_{4,k}\rangle$) from $\mathcal{S}_{\mathrm{REG}}$, the simulator $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{REG}}$ on input (reg.register.ini, $\langle sid, \mathcal{M}_j\rangle$, $\langle pk_{3,k}, pk_{4,k}\rangle$). When $\mathcal{F}_{\mathrm{REG}}$ outputs the message (reg.register.sim, $\langle sid, \mathcal{M}_j\rangle$, $\langle pk_{3,k}, pk_{4,k}\rangle$), $\mathcal{S}$ sends the message (reg.register.sim, $\langle sid, \mathcal{M}_j\rangle$, $\langle pk_{3,k}, pk_{4,k}\rangle$) to $\mathcal{S}_{\mathrm{REG}}$.
- On input the message (reg.register.rep, $\langle sid, \mathcal{M}_j\rangle$) from $\mathcal{S}_{\mathrm{REG}}$, the simulator $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{REG}}$ on input the message (reg.register.rep, $\langle sid, \mathcal{M}_j\rangle$). When $\mathcal{F}_{\mathrm{REG}}$ outputs the message (reg.register.end, $\langle sid, \mathcal{M}_j\rangle$), the simulator $\mathcal{S}$ sends the message (reg.register.end, $\langle sid, \mathcal{M}_j\rangle$) to $\mathcal{S}_{\mathrm{REG}}$.
- On input (reg.retrieve.ini, $\langle sid, \mathcal{M}_j\rangle$) from $\mathcal{S}_{\mathrm{REG}}$, $\mathcal{S}$ works as in the case where $\mathcal{U}$ is corrupt.
- On input (reg.retrieve.rep, $\langle sid, \mathcal{M}_j\rangle$, $ssid$) from $\mathcal{S}_{\mathrm{REG}}$, $\mathcal{S}$ works as in the case where $\mathcal{U}$ is corrupt.
- On input (smt.send.ini, sid_{SMT}, $\langle \mathcal{U}_i, bp, d, c, t, s\rangle$) from $\mathcal{S}_{\mathrm{SMT}}$, the simulator $\mathcal{S}$ checks that $sid_{\mathrm{SMT}} = (\mathcal{M}_j, \mathcal{U}_i, sid)$ where $\mathcal{M}_j$ is a corrupt meter. $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{SMT}}$ on input the message (smt.send.ini, sid_{SMT}, $\langle \mathcal{U}_i, bp, d, c, t, s\rangle$). When $\mathcal{F}_{\mathrm{SMT}}$ outputs the message (smt.send.sim, sid_{SMT}, $ssid$, $l(\langle \mathcal{U}_i, bp, d, c, t, s\rangle)$), the simulator $\mathcal{S}$ forwards it to $\mathcal{S}_{\mathrm{SMT}}$.
- On input (bil.consumption.sim, sid, $ssid$, $\mathcal{M}_j, \mathcal{U}_i$) from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ works as in the case where $\mathcal{U}$ is corrupt.
- On input (bil.consumption.end, sid, $\mathcal{M}_j$, bp, c, t) from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ works as in the case where $\mathcal{U}$ is corrupt.
- On input (smt.send.ini, sid_{SMT}, $\langle \mathcal{U}_i, bp, ctm[bp, \mathcal{U}_i], s\rangle$) from $\mathcal{S}_{\mathrm{SMT}}$, the simulator $\mathcal{S}$ checks that $sid_{\mathrm{SMT}} = (\mathcal{M}_j, \mathcal{U}_i, sid)$ where $\mathcal{M}_j$ is a corrupt meter. The simulator $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{SMT}}$ on input the message (smt.send.ini, sid_{SMT}, $\langle \mathcal{U}_i, bp, ctm[bp, \mathcal{U}_i], s\rangle$). When $\mathcal{F}_{\mathrm{SMT}}$ outputs the message (smt.send.sim, sid_{SMT}, $ssid$, $l(\langle \mathcal{U}_i, bp, ctm[bp, \mathcal{U}_i], s\rangle)$), the simulator $\mathcal{S}$ forwards it to $\mathcal{S}_{\mathrm{SMT}}$.
- On input (bil.period.sim, sid, $ssid$, $\mathcal{M}_j, \mathcal{U}_i$) from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ works as in the case where $\mathcal{U}$ is corrupt.
- On input (bil.period.end, sid, bp, $\mathcal{M}_j$, $N[\mathcal{M}_j, bp]$) from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ works as in the case where $\mathcal{U}$ is corrupt.
- On input (bil.payment.sim, sid, $ssid$, $\mathcal{U}_i, \mathcal{P}$) from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ works as in the case where $\mathcal{V}$ or $\mathcal{U}$ acting as verifying parties are corrupt.
- On input (bil.payment.end, sid, $\mathcal{U}_i$, bp, $p[bp]$, $\mathcal{M}_{j_1}$, $N[\mathcal{M}_{j_1}, bp]$, $\ldots$, $\mathcal{M}_{j_m}$, $N[\mathcal{M}_{j_m}, bp]$) from $\mathcal{F}_{\mathrm{BIL}}$, $\mathcal{S}$ works as in the case where $\mathcal{V}$ or $\mathcal{U}$ acting as verifying parties are corrupt.
- On input (smt.send.rep, sid_{SMT}, $ssid$) from $\mathcal{S}_{\mathrm{SMT}}$, $\mathcal{S}$ proceeds as follows:
  - If a message (smt.send.sim, sid'_{SMT}, $ssid'$, $\ldots$) such that $(sid_{\mathrm{SMT}}, ssid) = (sid'_{\mathrm{SMT}}, ssid')$ was not sent to $\mathcal{S}_{\mathrm{SMT}}$, $\mathcal{S}$ ignores the message.
  - Else, if the message (smt.send.sim, sid_{SMT}, $ssid$, $\ldots$) was sent after receiving a message (smt.send.ini, sid_{SMT}, $\langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s\rangle$) from $\mathcal{S}_{\mathrm{SMT}}$, $\mathcal{S}$ works as in the case where $\mathcal{V}$ is corrupt.
  - Else, if the message (smt.send.sim, sid_{SMT}, $ssid$, $\ldots$) was sent after receiving a message (smt.send.ini, sid_{SMT}, $\langle \mathcal{U}_i, bp, d, c, t, s\rangle$) from $\mathcal{S}_{\mathrm{SMT}}$, $\mathcal{S}$ runs the corresponding instance of $\mathcal{F}_{\mathrm{SMT}}$ on input (smt.send.rep, sid_{SMT}, $ssid$). When $\mathcal{F}_{\mathrm{SMT}}$ sends (smt.send.end, sid_{SMT}, $\langle \mathcal{U}_i, bp, d, c, t, s\rangle$), $\mathcal{S}$ does nothing if there is not an instance of $\mathcal{F}_{\mathrm{REG}}$ that stores $\langle pk_{3,k}, pk_{4,k}\rangle$. ($\mathcal{S}$ checks that the meter identifier $\mathcal{M}_j$ in sid_{SMT} and the meter identifier contained in the session identifier $\langle sid, \mathcal{M}_j\rangle$ of $\mathcal{F}_{\mathrm{REG}}$ are the same.) $\mathcal{S}$ does nothing if $1 \neq \mathsf{VfSig}_3(pk_{3,k}, s, \langle \mathcal{U}_i, bp, d, c, t\rangle)$. $\mathcal{S}$ does nothing if there is a tuple $(sid, \mathcal{U}'_i, \mathcal{M}'_j, bp', ctm'[bp, \mathcal{M}_j])$ stored such that $\mathcal{U}'_i = \mathcal{U}_i$, $\mathcal{M}'_j = \mathcal{M}_j$ and $bp' = bp$. $\mathcal{S}$ does nothing if it stores a tuple $[\mathcal{U}'_i, \mathcal{M}'_j, bp', d', c, t, s]$ such that $\mathcal{U}_i = \mathcal{U}'_i$, $\mathcal{M}'_j = \mathcal{M}_j$, $bp' = bp'$ and $d' = d$. Otherwise $\mathcal{S}$ stores $[\mathcal{U}_i, \mathcal{M}_j, bp, d, c, t, s]$. $\mathcal{S}$ sends (bil.consumption.ini, $sid, \mathcal{U}_i, bp, c, t$) to $\mathcal{F}_{\mathrm{BIL}}$. When $\mathcal{F}_{\mathrm{BIL}}$ outputs (bil.consumption.sim, sid, $ssid$, $\mathcal{M}_j, \mathcal{U}_i$), $\mathcal{S}$ sends (bil.consumption.rep, sid, $ssid$) to $\mathcal{F}_{\mathrm{BIL}}$.
  - Else, if the message (smt.send.sim, sid_{SMT}, $ssid$, $\ldots$) was sent after receiving a message (smt.send.ini, sid_{SMT}, $\langle \mathcal{U}_i, bp, ctm[bp, \mathcal{U}_i], s\rangle$) from $\mathcal{S}_{\mathrm{SMT}}$, $\mathcal{S}$ runs the corresponding instance of $\mathcal{F}_{\mathrm{SMT}}$ on input (smt.send.rep, sid_{SMT}, $ssid$). When $\mathcal{F}_{\mathrm{SMT}}$ sends (smt.send.end, sid_{SMT}, $\langle \mathcal{U}_i, bp, ctm[bp, \mathcal{U}_i], s\rangle$), $\mathcal{S}$ does nothing if there is not an instance of $\mathcal{F}_{\mathrm{REG}}$ that stores $\langle pk_{3,k}, pk_{4,k}\rangle$. ($\mathcal{S}$ checks that the meter identifier $\mathcal{M}_j$ in sid_{SMT} and the meter identifier contained in the session identifier $\langle sid, \mathcal{M}_j\rangle$ of $\mathcal{F}_{\mathrm{REG}}$ are the same.) $\mathcal{S}$ does nothing if $1 \neq \mathsf{VfSig}_4(pk_{4,k}, s, \langle \mathcal{U}_i, bp, ctm[bp, \mathcal{M}_j]\rangle)$. $\mathcal{S}$ does nothing if there is a tuple $(sid, \mathcal{U}'_i, \mathcal{M}'_j, bp', ctm'[bp, \mathcal{M}_j])$ stored such that $\mathcal{U}'_i = \mathcal{U}_i$, $\mathcal{M}'_j = \mathcal{M}_j$ and $bp' = bp$. $\mathcal{S}$ does nothing if the number of tuples $[\mathcal{U}'_i, \mathcal{M}'_j, bp', d, c, t, s]$ stored such that $\mathcal{U}'_i = \mathcal{U}_i$, $\mathcal{M}'_j = \mathcal{M}_j$ and $bp' = bp$ is different from $ctm[bp, \mathcal{M}_j]$. $\mathcal{S}$ also aborts if, from $d = 1$ to $ctm[bp, \mathcal{M}_j]$, $\mathcal{S}$ cannot find a tuple $[\mathcal{U}'_i, \mathcal{M}'_j, bp', ctm'[bp, \mathcal{M}_j], c, t, s]$ stored such that $\mathcal{U}'_i = \mathcal{U}_i$, $\mathcal{M}'_j = \mathcal{M}_j$ and $bp' = bp$ and $d = ctm'[bp, \mathcal{M}_j]$. Otherwise $\mathcal{S}$ stores $(sid, \mathcal{U}_i, \mathcal{M}_j, bp, ctm[bp, \mathcal{M}_j], s)$. $\mathcal{S}$ sends (bil.period.ini, $sid, \mathcal{U}_i, bp$) to $\mathcal{F}_{\mathrm{BIL}}$. When $\mathcal{F}_{\mathrm{BIL}}$ outputs (bil.period.sim, sid, $ssid$, $\mathcal{M}_j, \mathcal{U}_i$), $\mathcal{S}$ sends (bil.period.rep, sid, $ssid$) to $\mathcal{F}_{\mathrm{BIL}}$.
  - Else, if the message (smt.send.sim, sid_{SMT}, $ssid$, $\ldots$) was sent after receiving a message (bil.payment.sim, sid, $ssid, \mathcal{U}_i, \mathcal{P}$) from $\mathcal{F}_{\mathrm{BIL}}$, if $(ssid, p, \langle \mathcal{M}_{j_k}, ctm[bp, \mathcal{M}_{j_k}]\rangle_{k=1}^{m})$ is stored, $\mathcal{S}$ deletes that tuple and sends (bil.payment.rep, sid, $ssid$, p, $\langle \mathcal{M}_{j_k}, ctm[bp, \mathcal{M}_{j_k}]\rangle_{k=1}^{m}$) to $\mathcal{F}_{\mathrm{BIL}}$, else $\mathcal{S}$ sends the message (bil.payment.rep, sid, $ssid$) to the functionality $\mathcal{F}_{\mathrm{BIL}}$.

Fig. 9 Simulator $\mathcal{S}$: case $\mathcal{V}, \mathcal{U}$ and $\mathcal{M}$ corrupt (I)

Simulator $\mathcal{S}$: case $\mathcal{V}, \mathcal{U}$ and $\mathcal{M}$ corrupt (II)

- On input ($\mathsf{smt.send.ini}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$) from $\mathcal{S}_{\mathrm{SMT}}$, $\mathcal{S}$ proceeds as follows.
  - $\mathcal{S}$ ignores the message if pk_1 or pk_2 are not stored, or if $pk_{3,k}$ or $pk_{4,k}$ are not stored for any $\mathcal{M}_{j_k}$ (for $k = 1$ to m), or if $\boxed{\text{POL: } par_p,}$ par_c and par_{pk} are not stored.
  - $\mathcal{S}$ parses sid_{SMT} as $(\mathcal{U}_i, \mathcal{P}, sid)$ and checks that $\mathcal{U}_i$ is corrupt.
  - $\mathcal{S}$ checks if the values $\boxed{\text{POL: } par_p,}$ par_c, pk_1, $pk_{3,k}$, and $pk_{4,k}$ stored are equal to those in the instance $ins = (\boxed{\text{POL: } par_p,} par_c, pk_1, \mathcal{U}_i, com, bp, [pk_{3,k}, pk_{4,k}, ctm[bp, \mathcal{M}_{j_k}]]_{k=1}^{m})$. If not, $\mathcal{S}$ ignores the message. $\mathcal{S}$ also checks that, for $k = 1$ to m, the instance includes a counter $ctm[bp, \mathcal{M}_{j_k}]$ of meter readings and that the proof proves possession of $ctm[bp, \mathcal{M}_{j_k}]$ meter readings numbered from 1 to $ctm[bp, \mathcal{M}_{j_k}]$.
  - $\mathcal{S}$ ignores the message if $1 \neq \mathsf{VfCom}(par_c, com, p, open)$.
  - $\mathcal{S}$ ignores the message if $1 \neq \mathsf{VfSig}_2(pk_2, s, \langle bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}\rangle)$.
  - $\mathcal{S}$ ignores the message if $1 \neq \mathsf{PKVerify}(par_{pk}, ins, \pi)$.
  - $\mathcal{S}$ aborts if, for $k = 1$ to m, $\mathcal{M}_{j_k}$ is honest and the number of meter readings $ctm[bp, \mathcal{M}_{j_k}]$ contained in the instance ins is not the one sent to the adversary by $\mathcal{M}_{j_k}$ at that billing period.
  - $\mathcal{S}$ sends ($\mathsf{bil.payment.ini}$, $sid, \mathcal{P}, bp$) to $\mathcal{F}_{\mathrm{BIL}}$. When $\mathcal{F}_{\mathrm{BIL}}$ sends ($\mathsf{bil.payment.sim}$, $sid, ssid, \mathcal{U}_i, \mathcal{P}$), $\mathcal{S}$ stores ($ssid, p, \langle \mathcal{M}_{j_k}, ctm[bp, \mathcal{M}_{j_k}]\rangle_{k=1}^{m}$) and sends ($\mathsf{smt.send.sim}$, sid_{SMT}, $ssid$, $l(\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle)$) to $\mathcal{S}_{\mathrm{SMT}}$.

Fig. 10 Simulator $\mathcal{S}$: case $\mathcal{V}, \mathcal{U}$ and $\mathcal{M}$ corrupt (II)

edge scheme ($\mathsf{PKSetup}, \mathsf{PKProve}, \mathsf{PKVerify}$*) is zero-knowledge and extractable and the signature scheme (*$\mathsf{KeyGen}_4, \mathsf{Sign}_4, \mathsf{VfSig}_4$*) is existentially unforgeable.*

Proof We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish between the ensemble $\mathrm{REAL}_{\mathrm{BIL},\mathcal{A},\mathcal{Z}}$ and the ensemble $\mathrm{IDEAL}_{\mathcal{F}_{\mathrm{BIL}},\mathcal{S},\mathcal{Z}}$ with non-negligible probability. We denote by Pr [**Game** i] the probability that the environment distinguishes **Game** i from the real-world protocol.

Game 0 This game corresponds to the execution of the real-world protocol. Therefore, Pr [**Game** 0] = 0.

Game 1 **Game** 1 follows **Game** 0, except that **Game** 1 computes the parameters par_{pk} by running $(par_{pk}, td_e) \leftarrow \mathcal{E}_1(1^k)$. **Game** 2 stores td_e. The extraction property ensures that the parameters par_{pk} output by $\mathcal{E}_1(1^k)$ is indistinguishable from those output by $\mathsf{PKSetup}$. Therefore, $|\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{ex-nipk}}$.

Game 2 **Game** 2 follows **Game** 1, except that, when the adversary sends a message ($\mathsf{smt.send.ini}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$), after verifying s, com and π, **Game** 2 runs $wit \leftarrow \mathcal{E}_2(par_{pk}, td_e, ins, \pi)$. **Game** 2 aborts if extraction fails. The extraction property ensures that extraction works with overwhelming probability. Therefore, $|\Pr[\textbf{Game } 2] - \Pr[\textbf{Game } 1]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{ex-nipk}}$.

Game 3 **Game** 3 follows **Game** 2, except that, when the adversary sends ($\mathsf{smt.send.ini}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$), after extracting the witness wit, **Game** 3 aborts if any of the signatures s_k in the witness wit signs a tuple $\langle bp, \mathcal{U}_i, ctm[bp, \mathcal{M}_{j_k}]\rangle$ such that the meter $\mathcal{M}_{j_k}$ is honest and no signature on that tuple was sent to the adversary. Thanks to the existential unforgeability of the signature scheme ($\mathsf{KeyGen}_4, \mathsf{Sign}_4, \mathsf{VfSig}_4$), **Game** 3 aborts with negligible probability. Therefore, $|\Pr[\textbf{Game } 3] - \Pr[\textbf{Game } 2]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{unf-sig}}$.

In **Game** 3, we have shown that $\mathcal{S}$ receives with negligible probability a message ($\mathsf{smt.send.ini}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$) that makes $\mathcal{S}$ abort.

Game 4 **Game** 4 follows **Game** 3, except that **Game** 4 computes par_{pk} by running $\mathcal{S}_1(1^k)$. **Game** 4 stores td_s. **Game** 4 does not run the extractor $\mathcal{E}_2$. The zero-knowledge property ensures that par_{pk} output by $\mathcal{S}_1$ is indistinguishable from those output by $\mathsf{PKSetup}$. Therefore, $|\Pr[\textbf{Game } 4] - \Pr[\textbf{Game } 3]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{zk-nipk}}$.

Game 5 **Game** 5 follows **Game** 4, except that, when an honest user sends ($\mathsf{smt.send.ini}$, sid_{SMT}, $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi\rangle$), **Game** 5 computes the proof π by running $\pi \leftarrow \mathcal{S}_2(par_{pk}, td_s, ins)$. The zero-knowledge property ensures that proofs π computed by algorithm $\mathcal{S}_2$ are indistinguishable from those output by $\mathsf{PKProve}$. Therefore, we have that $|\Pr[\textbf{Game } 5] - \Pr[\textbf{Game } 4]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{zk-nipk}}$.

The distribution of **Game** 5 is identical to that of our simulation.

4.6.4 Case $\mathcal{V}$ and $\mathcal{U}$ corrupt

We omit a formal proof of this case. We give a high-level description of the simulator.

bil.policy.$*$, bil.listmeters.$*$ For these interfaces, the simulator $\mathcal{S}$ proceeds as in the case where the provider, a subset of the users and a subset of the meters are corrupt.
bil.consumption.$*$, bil.period.$*$ For these interfaces, the simulator $\mathcal{S}$ proceeds as in the case where only a subset of users is corrupt.
bil.payment.$*$ In this interface, the simulator proceeds as in the case where the provider, a subset of the users and a subset of the meters are corrupt. The only difference is that, when a corrupt user sends a payment message, since now all the meters are honest, the simulator does not need to check whether the meters involved in a payment message are honest or not, and thus, the simulator always aborts if the payment message contains a signature on a billing period, user identifier and counter of meter readings such that no signature on that tuple was sent to the adversary while simulating the corresponding honest meter.

4.6.5 Case $\mathcal{U}$ and $\mathcal{M}$ corrupt

We omit a formal proof of this case. We give a high-level description of the simulator.

bil.policy.$*$, bil.listmeters.$*$ For these interfaces, the simulator $\mathcal{S}$ simulates the honest provider toward the adversary as in the case where only a subset of the users is corrupt.
bil.consumption.$*$, bil.period.$*$ For these interfaces, when an honest meter sends a meter reading or an end of billing period message to a corrupt user, the simulator $\mathcal{S}$ proceeds as in the case where only a subset of users is corrupt. When a corrupt meter sends a message to an honest user, the simulator $\mathcal{S}$ proceeds as described in the case where a subset of users, a subset of meters and the provider are corrupt.
bil.payment.$*$ In this interface, when an honest user sends a payment message to a corrupt verifying party, the simulator proceeds as described in the case where only the provider is corrupt. If a corrupt user sends a payment message to an honest verifying party, the simulator $\mathcal{S}$ distinguishes between two cases. If all the meters involved in the payment message are honest, the simulator proceeds as described in the case where only a subset of the users is corrupt. If any of those meters is corrupt, the simulator $\mathcal{S}$ proceeds in a similar way as the one described for the case in which a subset of the meters, a subset of the users and the provider are corrupt. The only difference is that $\mathcal{S}$ also aborts if the payment message contains a signature on a list of meters, user identifier and billing period that was not sent to the adversary. Therefore, security in this case also relies on the existentially unforgeability of the signature scheme (KeyGen_2, Sign_2, VfSig_2).

4.6.6 Case $\mathcal{V}$ and $\mathcal{M}$ corrupt

We omit a formal proof of this case. We give a high-level description of the simulator.

bil.policy.$*$, bil.listmeters.$*$ For these interfaces, the simulator $\mathcal{S}$ proceeds as in the case where only the provider is corrupt.
bil.consumption.$*$, bil.period.$*$ For these interfaces, when a corrupt meter sends a message to an honest user, the simulator $\mathcal{S}$ proceeds as described in the case where a subset of users, a subset of meters and the provider are corrupt.
bil.payment.$*$ In this interface, when an honest user sends a payment message to a corrupt verifying party, the simulator proceeds as described in the case where only the provider is corrupt.

4.6.7 Case $\mathcal{V}$ and $\mathcal{M}$ corrupt but collusion-free

In Sect. 4.6.6, we have argued that our protocol realizes $\mathcal{F}_{\mathrm{BIL}}$. However, in that corruption model, the adversary controls all the corrupt parties and is thus able to communicate information between them. Therefore, a corrupt meter can communicate the meter readings of an honest user to any corrupt party, which violates user privacy.

In a smart metering setting, it is useful to consider a corruption model where the meters and the provider (or other verifying parties) are corrupt, but do not have a side communication channel between them. In our protocol, the only way such adversarial parties would have to coordinate their actions and to disclose information between each other would be to construct a side channel through the user.

However, we can show that our protocol does not allow that and is collusion-free in the sense of [33]. The payment message sent by a user to a verifying party is $\langle p, open, com, bp, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, s, ins, \pi \rangle$. The values $(p, open, com, \pi)$ are computed by the user. The billing period bp, the meter identifiers $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$ and the signature s are sent by the provider. The instance ins is of the following form.

$$ins = (\boxed{par_p}, par_c, pk_1, \mathcal{U}_i, com, bp, [pk_{3,k}, pk_{4,k}, ctm[bp, \mathcal{M}_{j_k}]]_{k=1}^{m}).$$

Here, only the values $[pk_{3,k}, pk_{4,k}, ctm[bp, \mathcal{M}_{j_k}]]_{k=1}^{m}$ are generated by the meter. The public keys $pk_{3,k}$ and $pk_{4,k}$ can be generated at setup, before meter readings are output. The counter $ctm[bp, \mathcal{M}_{j_k}]$ must employ a unique representation for all the numbers in its domain, so that the meter is not able to use it to disclose any information to the verifying parties. To prevent a corrupt meter from manipulating the value of the counter to convey information, it is possible to enforce a constant number of meter readings from each meter in a payment message. When these conditions are met, our protocol avoids a collusion between corrupt meters and verifying parties that do not have a side communication channel.

5 Related work

To the best of our knowledge, currently deployed fine-grained billing protocols reveal meter readings to the service provider. In the case of smart metering, relevant standards that define communication protocols between meters and service providers include ANSI C12.18, C12.19 and C12.22, and the open smart grid protocol. We refer to [23] for a wider overview of communication protocols and standards applicable to the smart grid. In the case of electronic toll collection, the Decision 2009/750/EC, which defines the European Electronic Toll Service and its technical elements, requires location data to be reported to the service provider for the purpose of billing.

In the context of the smart grid, several papers analyze the types of personal information that can be inferred from power consumption data [35,44]. They show how to infer many intimate details of users' daily lives. In the context of location-based applications, the privacy threats related to disclosing location data have also been analyzed [18].

In order to protect privacy in smart metering applications, several approaches have been considered in the literature:

Regulations and codes of conduct We find tools to define and enforce privacy policies [45], privacy-friendly access control protocols to ensure that data are only accessed by authorized parties [8], and audit tools to verify that no inappropriate access has taken place [5]. For example, Kumari et al. [31] propose usage control mechanisms for data shared by smart meters. In addition, there are transparency-enhancing tools that help users to understand how data are collected, shared, stored, processed and analyzed [11,38], which can be applied to smart metering.

Variability reduction Data mining methods take advantage of the changes in power consumption in order to infer personal data. One approach to minimize this information leakage consists in installing a rechargeable battery on the user's side [27,46,50]. The rechargeable battery inputs power at an (ideally) constant rate and outputs it depending on the user's needs. Therefore, the provider's view is that of a user whose consumption of electricity does not vary. In practice, the privacy provided by this approach depends on the capacity of the battery. If the user consumption is lower (resp. greater) than the battery consumption, the battery must reduce (resp. increase) its consumption, and thus, in practice it is not possible to achieve a constant rate.

Anonymization Anonymization techniques allow the service provider to obtain meter data from users without being able to tell apart the meter readings that belong to each individual user [16,51]. This technique can be useful for applications such as forecasting, leak detection or flow monitoring, where knowing the consumption of each user may not be needed. For billing, Popa et al. [43] propose a protocol based on anonymization for electronic toll collection. Users send location data segments anonymously to a database. The provider computes the prices for each segment and sends them to the users. Each user employs the prices corresponding to their segments to compute the total bill and a proof of correctness of the bill calculation. The main problems of anonymization techniques are that they require anonymous communication channels and that they are vulnerable to deanonymization attacks [30].

Differential Privacy Differential privacy methods consist in adding noise to meter readings in such a way that the result of a statistical query on a database of meter readings does not reveal any information about individual meter readings [1,4]. Differential privacy methods are difficult to apply in the case of billing because adding noise to consumption measurements leads to an inaccurate bill. Nevertheless, differential privacy has been used together with a privacy-preserving billing protocol to hide from the provider the bill to be paid by adding positive noise to it, together with a rebate mechanism that ensures that users get back their excess payments [15].

Verifiable Computing Verifiable computing allows a client with limited resources to outsource the computation of a function to an untrusted worker in such a way that the client is able to verify the correctness of the computation [20]. The basic requirement is that the cost of verifying correctness is smaller than the cost of computing the function. Some schemes provide public verifiability [42], so that the verification can be performed by any party. The zero-knowledge property ensures that the worker can convince the client that it knows an input that fulfills some property, while the client does not learn further information on the input beyond what can be inferred from the result of the function [41].

Verifiable computing can be applied to our setting as follows. The provider acts as the client and outsources the computation of the tariff policy to the user. The user

inputs the meter readings from the meter, uses the zero-knowledge property to prove that they are signed by the meter, performs the computation of the tariff policy and reveals to the provider the result, along with a proof of correctness.

Although verifiable computing can in principle be applied to our setting, there are several shortcomings. First, the provider is not resource constrained; it in fact possesses more computation power than the user. So in our protocol we do not focus on saving provider's resources at the expense of the user. Furthermore, verifiable computing schemes have a costly preprocessing phase where the client computes an evaluation key, which is sent to the worker. The cost of this preprocessing phase is amortized after outsourcing the computation of the function several times. However, in smart metering applications, tariff policies change dynamically depending on the power generation cost. For example, in Spain, the tariff policy changes hourly[1]. Therefore, it may be possible that the preprocessing cost is not amortized.

Trusted party Meter readings are sent to a trusted party that keeps them secret and only reveals the results of the computations done on them. Bohli et al. [6] propose a solution where a trusted party aggregates meter readings and reveals the aggregate to the provider.

Secure Two-Party Computation In these protocols, two parties, each of them with a private input, wish to jointly compute a function of their inputs and learn the result without disclosing their private inputs. The feasibility of secure two-party computation for any function has been shown [52], and subsequent works improve the efficiency of computation [34,39] and minimize the number of communication rounds [25,26]. Recently, some protocols involve a costly preprocessing phase and an efficient online phase [13], while others distribute the workload asymmetrically between participants [12], like in server-aided secure computation [28], but as mentioned above this is not advisable in our setting.

Our protocol is a secure two-party computation optimized for the task of billing. In this setting, only the user has a private input (the meter readings signed by the meter), while the provider only needs to verify the result of the computation. This allows us to design a simple non-interactive protocol where the user performs the bill calculation locally and sends the result to the provider, along with a proof of correctness.

Secure Multi-Party Computation In this case, several parties, each of them with a private input, compute jointly the result of function on input their private inputs. Parties obtain the result of the computation, but they do not learn the private inputs of the other parties. The feasibility of secure multi-party computation has been shown [22].

In the context of smart metering, secure multi-party computation has been applied to reveal to the service provider the result of a function that takes in the meter readings of more than one user. Some works focus on revealing to the service provider the aggregate consumption of a group of users, for purposes such as fraud detection, statistics collection or demand management [14,19,32,48]. We note that [48] shows a two-party protocol for billing purposes, but it requires to perform all the computation inside the tamper-resistant meter and the class of tariff policies that it supports is very limited.

[1] http://moneysaverspain.com/electricity-bill-spain/

6 Conclusion

Privacy-preserving billing protocols allow users to calculate the total bill on input meter readings and prove to the service provider that the bill is correct without disclosing meter readings. They are useful to protect user privacy in any application that employs fine-grained billing, such as smart metering, electronic traffic pricing and road tolling. First, we have revisited the security model in [47] and we have proposed an ideal functionality for privacy-preserving billing where a meter can output meter readings to multiple users, and where a user receives meter readings from multiple meters. We have also proposed a protocol that realizes our ideal functionality and that, for tariff policies described by splines, improves the communication cost of the protocol in [47].

References

1. Acs, G., Castelluccia, C.: I have a dream!(differentially private smart metering). In: Filler, T., Pevný, T., Craver, S., Ker, A. (eds.) Information hiding, pp. 118–132. Springer, Berlin, Heidelberg (2011)
2. Anderson, R., Fuloria, S.: On the security economics of electricity metering. In: 9th Annual workshop on the economics of information security, WEIS 2010, Harvard University, Cambridge, MA, USA, 7–8 June 2010
3. Balasch, J., Rial, A., Troncoso, C., Preneel, B., Verbauwhede, I., Geuens, C.: Pretp: Privacy-preserving electronic toll pricing. In: USENIX Security Symposium, pp. 63–78. USENIX Association (2010)
4. Barthe, G., Danezis, G., Grégoire, B., Kunz, C., Zanella-Béguelin, S.: Verified computational differential privacy with applications to smart metering. In: 2013 IEEE 26th Computer Security Foundations Symposium (CSF), pp. 287–301. IEEE (2013)
5. Biswas, D., Niemi, V.: Transforming privacy policies to auditing specifications. In: 2011 IEEE 13th International Symposium on High-Assurance Systems Engineering (HASE), pp. 368–375. IEEE (2011)
6. Bohli, J.M., Sorge, C., Ugus, O.: A privacy model for smart metering. In: 2010 IEEE International Conference on Communications Workshops (ICC), pp. 1–5. IEEE (2010)

7. Bordoff, J., Noel, P.: Pay-as-you-drive Auto Insurance: A Simple Way to Reduce Driving-Related Harms and Increase Equity. Hamilton Project Discussion Paper (2008)
8. Byun, J.W., Li, N.: Purpose based access control for privacy protection in relational database systems. VLDB J. **17**(4), 603–619 (2008). doi:10.1007/s00778-006-0023-0
9. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: FOCS, pp. 136–145. IEEE Computer Society (2001)
10. Canetti, R.: Universally composable signature, certification, and authentication. In: 2004 Proceedings of the 17th IEEE Computer Security Foundations Workshop, pp. 219–233. IEEE (2004)
11. Cranor, L., Langheinrich, M., Marchiori, M., Presler-Marshall, M., Reagle, J.: The platform for privacy preferences 1.0 (p3p1. 0) specification. W3C Recomm. **16** (2002). https://www.w3.org/TR/P3P/
12. Damgård, I., Faust, S., Hazay, C.: Secure two-party computation with low communication. In: Cramer, R. (ed.) Theory of cryptography, pp. 54–74. Springer, Berlin, Heidelberg (2012)
13. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in cryptology-CRYPTO 2012, pp. 643–662. Springer, Berlin, Heidelberg (2012)
14. Danezis, G., Fournet, C., Kohlweiss, M., Zanella-Béguelin, S.: Smart meter aggregation via secret-sharing. In: Proceedings of the First ACM Workshop on Smart Energy Grid Security, pp. 75–80. ACM (2013)
15. Danezis, G., Kohlweiss, M., Rial, A.: Differentially private billing with rebates. In: Filler, T., Pevný, T., Craver, S., Ker, A. (eds.) Information hiding, pp. 148–162. Springer, Berlin, Heidelberg (2011)
16. Efthymiou, C., Kalogridis, G.: Smart grid privacy via anonymization of smart metering data. In: 2010 First IEEE International Conference on Smart Grid Communications (SmartGridComm), pp. 238–243. IEEE (2010)
17. Fournet, C., Kohlweiss, M., Danezis, G., Luo, Z.: Zql: a compiler for privacy-preserving data processing. In: 22nd USENIX Security Symposium (USENIX Security 13), Washington, DC, pp. 163–178 (2013)
18. Freudiger, J., Shokri, R., Hubaux, J.P.: Evaluating the privacy risk of location-based services. In: Blythe, J. (ed.) Financial cryptography and data security, pp. 31–46. Springer, Berlin, Heidelberg (2012)
19. Garcia, F.D., Jacobs, B.: Privacy-friendly energy-metering via homomorphic encryption. In: Cuellar. J., Lopez, J., Barthe, G., Pretschner. A. (eds.) Security and trust management, pp. 226–238. Springer, Berlin, Heidelberg (2011)
20. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: Rabin, T. (ed.) Advances in cryptology–CRYPTO 2010, pp. 465–482. Springer, Berlin, Heidelberg (2010)
21. Goldwasser, S., Micali, S., Rivest, R.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM J. Comput. **17**(2), 281–308 (1988)
22. Goldwasser, S., Micali, S., Wigderson, A.: How to play any mental game, or a completeness theorem for protocols with an honest majority. In: Proceedings of the Nienteenth Annual ACM STOC, vol. 87, pp. 218–229 (1987)
23. Gungor, V.C., Sahin, D., Kocak, T., Ergut, S., Buccella, C., Cecati, C., Hancke, G.P.: Smart grid technologies: communication technologies and standards. IEEE Trans. Ind. Inform. **7**(4), 529–539 (2011)
24. Hensher, D.A.: Electronic toll collection. Transp. Res. A: Gen. **25**(1), 9–16 (1991)
25. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Prabhakaran, M., Sahai, A.: Efficient non-interactive secure computation. In: Paterson, K.G. (ed.) Advances in Cryptology–EUROCRYPT 2011, pp. 406–425. Springer, Berlin, Heidelberg (2011)
26. Ishai, Y., Prabhakaran, M., Sahai, A.: Founding cryptography on oblivious transfer—efficiently. In: Wagner, D. (ed.) Advances in Cryptology–CRYPTO 2008, pp. 572–591. Springer, Berlin, Heidelberg (2008)
27. Kalogridis, G., Efthymiou, C., Denic, S.Z., Lewis, T.A., Cepeda, R.: Privacy for smart meters: towards undetectable appliance load signatures. In: 2010 First IEEE International Conference on Smart Grid Communications (SmartGridComm), pp. 232–237. IEEE (2010)
28. Kamara, S., Mohassel, P., Riva, B.: Salus: a system for server-aided secure function evaluation. In: Proceedings of the 2012 ACM conference on Computer and communications security, pp. 797–808. ACM (2012)
29. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: Abe, M. (ed.) ASIACRYPT, Lecture Notes in Computer Science, vol. 6477, pp. 177–194. Springer, New York (2010)
30. Krumm, J.: Inference attacks on location tracks. In: LaMarca, A., Langheinrich, M., Truong, K.N. (eds.) Pervasive computing, pp. 127–143. Springer, Berlin, Heidelberg (2007)
31. Kumari, P., Kelbert, F., Pretschner, A.: Data protection in heterogeneous distributed systems: a smart meter example. In: Proceedings of dependable software for critical infrastructures, Berlin, 6 October 2011
32. Kursawe, K., Danezis, G., Kohlweiss, M.: Privacy-friendly aggregation for the smart-grid. In: Fischer–Hübner, S., Hopper, N. (eds.) Privacy enhancing technologies, pp. 175–191. Springer, Berlin, Heidelberg (2011)
33. Lepinksi, M., Micali, S., et al.: Collusion-free protocols. In: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing, pp. 543–552. ACM (2005)
34. Lindell, Y., Pinkas, B.: Secure two-party computation via cut-and-choose oblivious transfer. J. Cryptol. **25**(4), 680–722 (2012)
35. Lisovich, M., Wicker, S.: Privacy concerns in upcoming residential and commercial demand-response systems. In: 2008 Clemson University Power Systems Conference. Clemson University (2008). http://www.truststc.org/pubs/332.html
36. Massoud Amin, S., Wollenberg, B.F.: Toward a smart grid: power delivery for the 21st century. IEEE Power Energy Mag. **3**(5), 34–41 (2005)
37. Meiklejohn, S., Mowery, K., Checkoway, S., Shacham, H.: The phantom tollbooth: Privacy-preserving electronic toll collection in the presence of driver collusion. In: USENIX Security Symposium, vol. 201 (2011)
38. Nguyen, D.H., Mynatt, E.D.: Privacy mirrors: understanding and shaping socio-technical ubiquitous computing systems (2002)
39. Nielsen, J.B., Orlandi, C.: Technical report, Georgia Institute of Technology
40. Ogden, K.: Privacy issues in electronic toll collection. Transp. Res. C: Emerg. Technol. **9**(2), 123–134 (2001)
41. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 238–252. IEEE (2013)
42. Parno, B., Raykova, M., Vaikuntanathan, V.: How to delegate and verify in public: verifiable computation from attribute-based encryption. In: Cramer, R. (ed.) Theory of cryptography, pp. 422–439. Springer, Berlin, Heidelberg (2012)
43. Popa, R.A., Balakrishnan, H., Blumberg, A.J.: Vpriv: Protecting privacy in location-based vehicular services. In: USENIX Security Symposium, pp. 335–350 (2009)
44. Quinn, E.L.: Privacy and the new energy infrastructure. Available at SSRN 1370731 (2009)
45. Quinn, E.L.: Smart metering and privacy: existing laws and competing policies. Available at SSRN 1462285 (2009). doi:10.2139/ssrn.1462285

46. Rajagopalan, S.R., Sankar, L., Mohajer, S., Poor, H.V.: Smart meter privacy: A utility-privacy framework. In: 2011 IEEE International Conference on Smart Grid Communications (SmartGridComm), pp. 190–195. IEEE (2011)
47. Rial, A., Danezis, G.: Privacy-preserving smart metering. In: Chen, Y., Vaidya, J. (eds.) WPES, pp. 49–60. ACM, New york (2011)
48. Thoma, C., Cui, T., Franchetti, F.: Secure multiparty computation based privacy preserving smart metering system. In: 2012 North American Power Symposium (NAPS), pp. 1–6. IEEE (2012)
49. Troncoso, C., Danezis, G., Kosta, E., Preneel, B.: Pripayd: privacy friendly pay-as-you-drive insurance. In: Ning, P., Yu, T. (eds.) WPES, pp. 99–107. ACM, New York (2007)
50. Varodayan, D., Khisti, A.: Smart meter privacy using a rechargeable battery: minimizing the rate of information leakage. In: 2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 1932–1935. IEEE (2011)
51. Wang, S., Cui, L., Que, J., Choi, D.H., Jiang, X., Cheng, S., Xie, L.: A randomized response model for privacy preserving smart metering. IEEE Trans. Smart Grid **3**(3), 1317–1324 (2012)
52. Yao, A.C.C.: How to generate and exchange secrets. In: 1986 27th Annual Symposium on Foundations of Computer Science, pp. 162–167. IEEE (1986)