# Are ideal functionalities really ideal?

Myrto Arapinis
University of Edinburgh

Véronique Cortier
Université de Lorraine, CNRS, Inria, LORIA

Hubert de Groote
ENS Paris-Saclay, Université Paris-Saclay

Charlie Jacomme
Université de Lorraine Inria, CNRS, LORIA

Steve Kremer
Université de Lorraine Inria, CNRS, LORIA

***Abstract*—Ideal functionalities are used to study increasingly complex protocols within the Universal Composability framework. However, such functionalities are often complex themselves, making it difficult to assess whether they truly fulfill their promises. In this paper, we present four attacks on functionalities from various applications (e-voting, SMPC, anonymous lotteries, and smart metering), demonstrating that they do not capture the intuitively expected properties.**

**We argue that ideal functionalities should not merely be justified secure at a high level but rigorously proven to be so. To this end, we propose a methodology that combines game-based proofs and computer-aided verification: ideal functionalities can in fact be treated as protocols, and one can use traditional game-based proofs to study them, where any game-based security property proven on the functionality does transfer to any protocol that realizes it. We also propose fixed versions of the ideal functionalities we studied, and formally define the security properties they should satisfy through a game. Finally, using SQUIRREL, a proof assistant for protocol security, we formally prove that the fixed functionalities verify the specified game-based security properties.**

## I. INTRODUCTION

Two main approaches are typically used to formally define the security of protocols in the computational setting: game-based definitions and ideal functionalities. Game-based definitions formalize specific security properties as a game that the adversary wins if they break the property. For example, to formalize anonymity, one can typically consider a game where the adversary tries to guess whether they are interacting with Alice or Bob. Such game-based definitions are convenient for conducting proofs by reduction, where the game is step-by-step transformed into the security assumptions. This approach is also useful for considering different corruption scenarios for different security properties. For instance, in the context of voting, a voter's personal device may be trusted for vote privacy but not for verifiability (ensuring that the correct vote is counted), as specified, *e.g.*, in the Swiss legal requirements [1].

A drawback of the game-based approach is that each desired property must be modeled individually as a game, potentially overlooking some interesting ones. In contrast, ideal functionalities capture a protocol's entire desired behavior within a single definition. If a protocol $\Pi$ can be shown to securely realize an ideal functionality $\mathcal{F}$, then $\Pi$ is at least as secure as $\mathcal{F}$. Moreover, when used in the Universal Composability (UC) framework [2], ideal functionalities can easily be composed, which allows for modular reasoning and proof reuse.

However, ideal functionalities are increasingly complex. This is because they are applied to complex security protocols, with rich interfaces, which must be reflected in the functionality's design. Additionally, functionalities need to account for the accepted information leakage or the protocol's limitations, as real-world protocols often have caveats. As a result, despite being "ideal", a functionality's description can span several pages (*e.g.*, [3], [4], [5], [6]).

It is therefore difficult to gain a clear understanding of the properties ensured by such complex ideal functionalities. Actually, in recent years, papers often include an informal argument for why the proposed ideal functionality is secure (*e.g.*, [7], [8], [9]). But, it is well-known that even very simple protocols can be subject to flaws discovered years later, as exemplified by the famous attack on Needham-Schroeder [10], [11]. Interestingly, such attacks are often simple, do not exploit the underlying cryptography, and only interfere with the protocol's logic. Ideal functionalities could also be vulnerable to similar logical attacks. Specifically, privacy properties seem hard to analyze without formal proof: even if an ideal functionality does not leak the parties' identities, an adversary might still infer who sent a message by observing different behaviors, as we demonstrate in this paper.

*Our contributions.* Our first contribution is the identification of flaws in existing ideal functionalities from the literature, across four application domains: voting [12], anonymous lotteries in blockchains [13], [14], smart metering [3], and secure multi-party computation (SMPC) with identifiable aborts [15], [16], [17]. In each case, we demonstrate that the ideal functionality does not satisfy the claimed security property:

- The seminal ideal functionality for voting [12] is simple and natural. Yet, we show that it can be realized by insecure protocols because the ideal functionality hides when a voter is denied the ability to vote by the adversary. This makes the ideal functionality easier to simulate, but such behaviors typically do not happen in real protocols.
- The ideal functionality modeling an anonymous lottery in [13] exemplifies the difficulty of analyzing privacy properties. Although the functionality does not explicitly reveal who was selected by the lottery, an adversary can infer this by observing the functionality's response to different (adversarially crafted) requests.
- In the context of secure billing, both the protocol and the

This work benefited from funding managed by the French National Research Agency under the France 2030 programme with the reference ANR-22-PECY-0006.

functionality are complex, designed to account for multiple meters, setup, etc. This renders the functionality difficult to analyze "by hand". We show that an attacker controlling at least one meter can manipulate the paid price, in particular excluding the price derived from the honest meters' reading.
- Finally, we study two variants of functionalities in the context of SMPC with identifiable aborts. The objective being that in case a party decides to abort the MPC protocol, either everyone still receives the result of the computation, or one of the faulty parties is publicly identified. The difficulty lies in this property only being "eventually" true, after a couple of further rounds of interactions. We demonstrate that several functionalities [15], [16] actually allow an adversary to abort without any honest party being informed.

Note that our findings do not imply any flaw in the proofs of secure realization, nor do they constitue actual attacks against the protocols under study. Instead, we are left in a situation where a protocol has been shown to realize an insecure ideal functionality, thus its security can no longer be guaranteed.

To avoid such an embarrassing situation, we advocate treating an ideal functionality as a protocol and analyzing it as such. Hence, for each ideal functionality we have considered, we propose a game-based definition that reflects the desired property. This might seem surprising, as an ideal functionality is often seen as a security definition itself. We argue, however, that this step is both necessary to assess the security of the ideal functionality and complementary to the UC approach: the ideal functionality still captures more properties than those expressed as game-based definitions, and the UC approach remains much more amenable to secure composition. While designing a faithful game-based definition may be as hard as writing a faithful ideal functionality, it allows for cross-checking the security of the ideal functionality with a different mindset, thereby providing a higher level of confidence in the functionality. We stress that we do not argue that game-based definitions should replace ideal functionalities. Game-based definitions are also subtle to define as illustrated by the variants of IND-CCA [18], [19] or e-voting privacy [20] definitions. However, verifying that an ideal functionality verifies some standard or newly defined game-based definition can only increase trust in the functionality, and is therefore a valuable validation method. Naturally, whenever an ideal functionality $\mathcal{F}$ satisfies a game-based property, any protocol that UC-realizes $\mathcal{F}$ also satisfies the property. We formally prove this folklore result for a general class of game-based properties.

Our second contribution consists of formal proofs that the previously examined ideal functionalities (possibly after some fix) satisfy their respective game-based properties. Thanks to the high level of abstraction of an ideal functionality, such proofs no longer involve the security of underlying cryptographic primitives. However, these proofs can still be cumbersome, as they require keeping track of lists, states, etc. To provide a high level of confidence in our security proofs, we use SQUIRREL [21], an interactive theorem prover that allows us to prove the security of protocols formalized as security games. It internally employs a logic that supports reasoning about what an attacker cannot do and includes a library of sound rules. SQUIRREL is well-suited for protocols with states or interleaving, which makes it a natural candidate for our study. This was confirmed by our ability to get formal security proofs for the ideal functionalities across all our case-studies.

*Related work:* We are not the first to identify the challenges in devising abstract, ideal-process-based definitions that truly capture UC composable security notions. Quoting Canetti [22] regarding a signature functionality, "*the formalization in [23], as well as the subsequent re-formulations of $\mathcal{F}_{\text{SIG}}$ in [24], [25], [26], contain a number of technical flaws that make the claim of equivalence to CMA security incorrect.*" Therefore, Canetti [22] directly justifies the soundness of its newly proposed ideal signature functionality by showing its equivalence with the well-established standalone security notion of CMA-security [27]. A similar approach and result were explored for encryption in [28]. In that work, the authors propose Replayable CCA (RCCA) security, a relaxed variant of traditional Chosen-Ciphertext Attack (CCA) security and show its equivalence with two game-based definitions, Indistinguishability-based RCCA (IND-RCCA) and Non-Malleability-based RCCA (NM-RCCA), for schemes operating over large message domains, which encompasses most practical encryption schemes.

Similar to our approach, the aforementioned works rigorously analyze the security guarantees of UC ideal functionalities. They primarily achieve this by proving equivalences or strong implications between these composable, ideal-process-based notions and well-established standalone definitions.

Work also exists on computer-aided proofs in the UC setting. Delaune *et al.* [29], and subsequently Böhl and Unruh [30], propose symbolic versions of the UC framework in the applied pi calculus. Their goal was to leverage: *(i)* the advantages of the UC framework (specification through ideal functionalities and composability) to the symbolic setting, and *(ii)* the advantages of the symbolic setting, by tool-assisting the UC-realization proof using symbolic provers such as ProVerif [31]. Canetti and Herzog [32] also use the UC framework to achieve composability in the symbolic setting, through a *computational soundness* result. Finally, in [33], Canetti *et al.* present a methodology for certifying UC proofs in the EasyCrypt proof assistant. We note that this line of research complements ours: the aforementioned tools can indeed be used to prove that a protocol realizes some ideal functionality, but these works do not aim at verifying whether the ideal functionality indeed provides the expected security guarantees. We fill this gap here, by defining how to formally prove that ideal functionalities do indeed satisfy desirable security properties.

## II. BACKGROUND ON UC

The Universal Composability (UC) framework, introduced by Canetti in [2], enables formal security arguments for cryptographic protocols that continue to hold even under concurrent execution. The UC framework follows the *real/ideal world*

paradigm, which compares the execution of a concrete ("real-world") protocol to an idealised execution involving an *ideal functionality* capturing the intended secure behavior.

*Model of computation:* Within the Universal Composability (UC) framework, all participants are modeled as probabilistic polynomial-time interactive Turing machines instances (ITIs)[1]. The main entities are as follows:

- Protocol parties: If honest, these ITIs execute the protocol; if corrupted, they follow the adversary's instructions.
- The Environment ($\mathcal{Z}$): This ITI provides inputs to and receives outputs from the protocol parties and the adversary. The environment also schedules the execution.
- The Adversary ($\mathcal{A}$): This ITI can corrupt partiesand interacts with $\mathcal{Z}$. In the ideal world, the adversary is often referred to as the Simulator ($\mathcal{S}$).
- The Ideal functionality ($\mathcal{F}$): This ITI defines the desired behavior of the protocol and acts as a trusted party in the ideal execution. Any inherent information leakage (for example, the length of a ciphertext in an ideally secure channel) is formally modeled through the interaction between $\mathcal{F}$ and the ideal adversary $\mathcal{S}$.

*Model of communication:* In the UC framework, the interaction and information exchange among ITIs are coordinated through a system of specialized tapes. They define the primary interfaces through which ITIs process external information:

- Input tape: Through this tape an ITI receives input from a calling ITI.
- Subroutine-output tape: This tape is the dedicated mechanism through which a calling ITI receives the computational outputs from a subroutine ITI it has invoked.
- Backdoor tape: This specialized tape provides a distinct interface for an ITI to receive direct messages originating from the adversary.

Together, these receiving tapes delineate an ITI's interface.

*Model of Execution :* The UC framework adopts a single-threaded model of execution, with only one ITI allowed to be active at any given moment. The ITI scheduled for the next activation is determined by the currently active ITI. An activation of an ITI $M$, is defined as the sequence of computation steps that begins when $M$ is in an active configuration and continues until $M$ reaches an inactive configuration. Then, the activation is complete, and $M$ pauses while awaiting its next activation. If $M$ enters a special halt state, it is permanently deactivated; in all subsequent attempts at activation, it will immediately reset its activation bit, effectively terminating its role in the execution. The primary mechanism that determines the sequence of activations is the external-write instruction. Specifically, when a source ITI $M$, executes an external-write operation targeting a receiving tape (i.e., input, backdoor, or subroutine-output) of a target ITI $M'$, the target ITI $M'$ is designated as the next machine to be activated.

A common and convenient abstraction in network specifications is broadcast communication (send-to-all). The UC framework allows handling this operation while strictly adhering to its single-threaded execution model by relying on the ideal functionality's internal state and programmed logic [34]. When $\mathcal{F}$ receives a command that results in a broadcast, it performs the first external-write (activating the first recipient) and critically stores the message and the list of remaining recipients in its private state, marking its status as "broadcast in progress." When $\mathcal{F}$ is subsequently activated, its program logic is specified to prioritize this internal state: if the state indicates a pending broadcast, $\mathcal{F}$ ignores all new external commands. Instead, it executes the next sequential external-write to the next recipient. This sequence of activations and prioritized processing continues across multiple sequential activations of $\mathcal{F}$ until the message has been written on all target parties' tapes. Only after the final write does $\mathcal{F}$ clear its "broadcast in progress" state and resume processing new commands. Consequently, high-level commands like "send to all parties" found in ideal functionality specifications are often understood in the literature as syntactic sugar—a convenient abstraction for the underlying sequential, state-dependent activation encoding just described. This technique to enforce sequential atomicity also allows the convenient abstraction of other complex scheduling requirements, such as synchronous communication.

*Modelling party corruption :* The UC framework includes a mechanism for modelling party corruption to capture a variety of threat models, such as secrecy against eavesdroppers and resilience to adversarial behavior.

- Corruption Mechanism: The adversary $\mathcal{A}$ corrupts an ITI by writing $(\mathsf{corrupt}, cp)$ on the ITI's backdoor tape. The parameter $cp$ specifies the type of corruption (e.g., crash or malicious).
- Behavior Change: Upon corruption, the ITI's behavior (its code) is updated to follow the adversary's instructions. A corrupted party essentially becomes part of the adversary.
- Environment Awareness: A key aspect of the UC model is managing the extent to which the Environment $\mathcal{Z}$ is made aware of the corruption. This is controlled by an identity masking function $f$. This function ensures that the simulation-based security definition remains meaningful limiting the information $\mathcal{Z}$ obtains about the specific subroutine structure of the protocol being analyzed.

*Real vs. Ideal world paradigm:* Security is defined via a *simulation-based approach* which compares the execution of the real-world protocol with the one of an idealized world. In the *real world*, honest parties execute the actual protocol $\Pi$ in the presence of adversary $\mathcal{A}$ that controls corrupted parties. In the *ideal world*, parties interact only with the ideal functionality $\mathcal{F}$, and adversarial behavior is modeled by a simulator $\mathcal{S}$ that interacts with the environment $\mathcal{Z}$ and $\mathcal{F}$. In more details, the execution in each world proceeds as follows.

- **Ideal World:** In this setting, the *environment* $\mathcal{Z}$ manages the execution and provides inputs to the protocol parties. These parties are *dummies*; they simply forward all received inputs to the *ideal functionality* $\mathcal{F}$. The simulator $\mathcal{S}$ interacts with the environment $\mathcal{Z}$ and controls all corrupted parties in this

[1]An ITI $M = (\mu, id)$ consists of its code $\mu$ specified as an Interactive Turing Machine (ITM) and a unique identity $id$.

ideal scenario. We denote the output of $\mathcal{Z}$ at the end of the ideal world execution as $\text{EXEC}_{\mathcal{S},\mathcal{Z}}(\mathcal{F})$.

- **Real World:** Here, the environment $\mathcal{Z}$ provides inputs to and schedules the protocol parties, who then execute the protocol $\Pi$ according to its specifications. This execution takes place in the presence of the real adversary $\mathcal{A}$, which interacts with $\mathcal{Z}$ and has the ability to corrupt a certain number of parties. We denote the output of $\mathcal{Z}$ at the end of the real world execution as $\text{EXEC}_{\mathcal{A},\mathcal{Z}}(\Pi)$.

**Definition 1** (UC Security)**.** A protocol $\Pi$ is said to *UC-realize* the ideal functionality $\mathcal{F}$, if for every real world adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$, such that for every environment $\mathcal{Z}$, the distributions $\text{EXEC}_{\mathcal{A},\mathcal{Z}}(\Pi)$ and $\text{EXEC}_{\mathcal{S},\mathcal{Z}}(\mathcal{F})$ are computationally indistinguishable:

$$\forall \mathcal{A}\ \exists \mathcal{S}\ \forall \mathcal{Z},\ \text{EXEC}_{\mathcal{A},\mathcal{Z}}(\Pi) \simeq \text{EXEC}_{\mathcal{S},\mathcal{Z}}(\mathcal{F}).$$

A core property of UC security is its support for modular composition. With the universal composition theorem [2], if a protocol $\Pi$ UC-realises $\mathcal{F}$, then any protocol $\rho$ that uses $\mathcal{F}$ as a subroutine remains secure when $\mathcal{F}$ is replaced by $\Pi$. Security is then preserved under concurrent and nested executions.

## III. Methodology

### A. Our approach

The key step when studying the security of a protocol is to show that it UC-realizes the corresponding ideal functionality. Yet, the ideal functionality is not always "obviously" secure, and potential flaws can be overlooked, as exemplified in later sections. Hence, it is necessary to study the security of the functionality itself. For this, we propose a three-step approach:

1) define the security properties that the ideal functionality should satisfy (such as vote privacy, or correct billing, etc) through a game-based definition;
2) show that these security properties can be transferred to the initial protocol. This should be immediate thanks to the UC-realization and our transfer result, (Theorem 1) in Section III-B;
3) *prove* that the ideal functionality indeed satisfies the security game(s). In this paper, we propose a methodology for such proofs, based on SQUIRREL [21], as explained in Sections III-C and VI.

### B. Transfer result

We show a folklore result, namely that security properties satisfied by a functionality $\mathcal{F}$ are inherited by any protocol $\Pi$ that realizes $\mathcal{F}$. This can be formally proven for security properties that are defined using a specific type of cryptographic games, which we call UC-compatible security games.

*UC-compatible security games:* Game-based security definitions are formulated as reachability or indistinguishability games that define a Challenger $\mathcal{C}$, who controls the honest parties, and its interactions with the Adversary $\mathcal{A}$, who controls the corrupted ones. In this work, we focus on security games that are sound with respect to UC-realization. Consequently, we need to restrict the type of security games we consider, and we call these *UC-compatible games*. In such games, $\mathcal{C}$ has black-box access to the honest parties, i.e., it interacts with the honest parties solely through their environment-facing interfaces. Formally, a UC-compatible game is any probabilistic polynomial-time ITM that can interact as the environment with the protocol's parties and the adversary. Figure 1 illustrates the class of UC-compatible security games. For this class of games, the transfer result immediately follows.

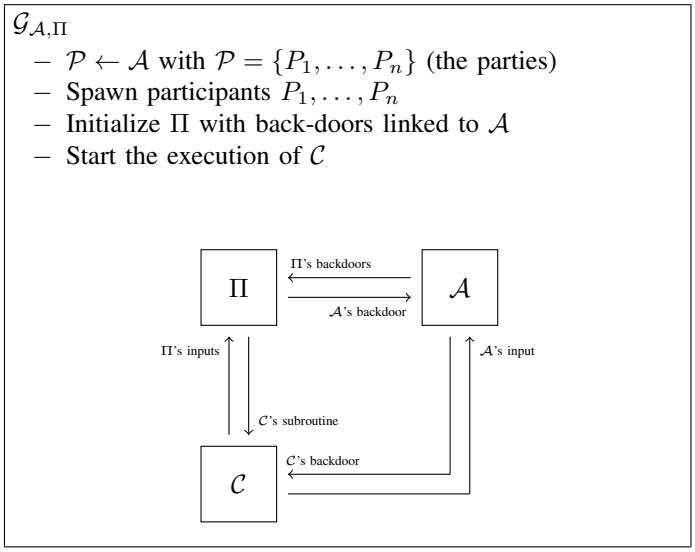


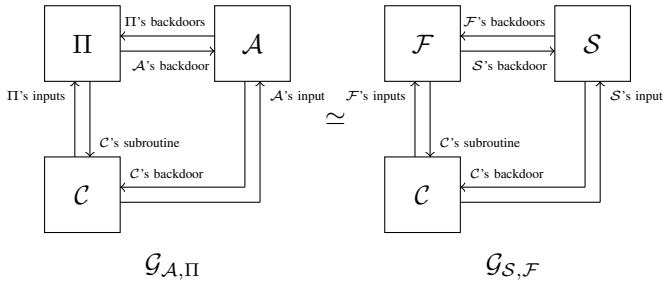

Figure 1: The challenger exposes a simulation of the honest parties of $\Pi$ to $\mathcal{A}$. $\mathcal{A}$ can do external writes on the backdoor tapes of protocol parties $P_1, \ldots, P_n$. Protocol parties $P_1, \ldots, P_n$ can do external writes on the backdoor tape of $\mathcal{A}$. The challenger is the environment in the eyes of $\mathcal{A}$ and $P_1, \ldots, P_n$, i.e. it can write on their input tapes. $\mathcal{A}$ and $P_1, \ldots, P_n$ return messages to $\mathcal{C}$'s subroutine output tape.

**Theorem 1** (Transfer)**.** Let $\Pi$ be a UC protocol, $\mathcal{F}$ an UC ideal functionality and $\mathcal{G}_{\mathcal{A},\Pi}$ a UC-compatible security game. If:

1) $\forall \mathcal{A}\ \exists \mathcal{S}\ \forall \mathcal{Z},\ \text{EXEC}_{\mathcal{A},\mathcal{Z}}(\Pi) \simeq \text{EXEC}_{\mathcal{S},\mathcal{Z}}(\mathcal{F})$ (UC realisation)
2) $\forall \mathcal{S},\ \Pr[\mathcal{G}_{\mathcal{S},\mathcal{F}} = 1] \leq \varepsilon(\lambda)$ ($\mathcal{F}$ is secure) for some negligible function $\varepsilon$

then $\Pi$ is also secure: $\forall \mathcal{A},\ \Pr[\mathcal{G}_{\mathcal{A},\Pi} = 1] \leq \varepsilon'(\lambda)$ for some negligible function $\varepsilon'$.

*Proof.* Let $\mathcal{A}$ be an ITM. By definition, the game $\mathcal{G}_{\mathcal{A},\Pi}$ (respectively $\mathcal{G}_{\mathcal{S},\mathcal{F}}$) can be interpreted as a UC execution with specific environment $\mathcal{C}$ that corresponds to the game. That is, $\mathcal{G}_{\mathcal{A},\Pi} = \text{EXEC}_{\mathcal{A},\mathcal{C}}(\Pi)$ and $\mathcal{G}_{\mathcal{S},\mathcal{F}} = \text{EXEC}_{\mathcal{S},\mathcal{C}}(\mathcal{F})$. Now, thanks to UC realisation, we know that there exists a simulator $\mathcal{S}$ such that $\forall \mathcal{Z}\ \text{EXEC}_{\mathcal{A},\mathcal{Z}}(\Pi) \simeq \text{EXEC}_{\mathcal{S},\mathcal{Z}}(\mathcal{F})$. So we can trivially conclude that there exists $\mathcal{S}$ such that $\mathcal{G}_{\mathcal{A},\Pi} \simeq \mathcal{G}_{\mathcal{S},\mathcal{F}}$




$\mathcal{G}_{\mathcal{A},\Pi}$ $\mathcal{G}_{\mathcal{S},\mathcal{F}}$

Since $\Pr[\mathcal{G}_{\mathcal{S},\mathcal{F}} = 1] \leq \varepsilon(\lambda)$ ($\mathcal{F}$ is secure), we immediately deduce that $\Pr[\mathcal{G}_{\mathcal{A},\Pi} = 1] \leq \varepsilon'(\lambda)$, and thus that $\Pi$ is secure. □

This theorem formally validates our approach, we can use games to verify the security guarantees of an ideal functionality.

Our soundness result applies to any type of corruption. However, UC realization is proven with respect to a specific corruption model (including an identity masking function $f$). Consequently, soundness is meaningful only for games that do not assume a higher level of leakage to $\mathcal{Z}$ than what $f$ specifies. In practice, this is rarely a restriction, as most of the UC literature concerning static and adaptive Byzantine corruption implicitly assumes an identity masking function that only reveals the identities of all corrupted ITIs to $\mathcal{Z}$, though not their internal code structure, also termed as *pid-wise corruption* [35], [36], [37], [9]. Similarly, most security games in the literature assume that the challenger only knows the identities of the corrupted parties.

**Limitation of the theorem .** The UC realization property provides an indistinguishability result up to a negligible advantage. Hence, properties that require perfect indistinguishability with a null advantage do not transfer. Typically, if one proves that an ideal functionality verifies some correctness property with probability 1, this does only transfer to its realizations with overwhelming probability. Note that this is not necessarily an indication of insecurity, but if a protocol is expected to satisfy a property with probability 1, this must be verified outside of the UC framework as an additional property.

Further, not all security properties can be specified as a security game. For instance, liveness properties are notoriously difficult to express generally and are not defined as standard security games [38]. Hence, Theorem 1 does not address whether a liveness property can be transferred. Characterizing the set of properties that can be transferred from a functionality to its (secure) realizations is out of the scope of this paper.

### C. Mechanized formal proofs in SQUIRREL

To get strong guarantees over ideal functionalities using our methodology, we leverage computer-aided cryptography. Ideal functionalities, in essence, are protocols that rely on global shared states rather than cryptography. Interestingly, we can thus use interchangeably tools that operate in the symbolic model like TAMARIN [39] or PROVERIF [40], or those that operate in the computational model, such as SQUIRREL [41], CRYPTOVERIF [42] or EASYCRYPT [43]. These provers primarily differ in their modeling of cryptography, which is irrelevant here since most ideal functionalities do not rely on cryptographic operations.

While various tools can be applied in specific scenarios, our aim is to demonstrate the versatility of our approach with diverse examples. Generally, ideal functionalities can involve arbitrary complex computations and numerous distinct procedures, as illustrated by Section V-C. Among the available tools, SQUIRREL best satisfies all our criteria: it has good built-in support for highly stateful protocols, that CRYPTOVERIF cannot handle; it can easily reason over protocols with complex interleaving of actions, for which EASYCRYPT does not scale well; and it can easily abstract away complex computations through dedicated logical axioms, which TAMARIN and PROVERIF find challenging to manage.

**Background on SQUIRREL.** SQUIRREL is a mechanized prover dedicated to cryptography, that models and proves security using a higher-order logic. While fully reasoning within a logic and its corresponding proof system, SQUIRREL provides computational guarantees, that is, it proves that some property is satisfied with overwhelming probability, against any polynomial-time probabilistic Turing machine. SQUIRREL models protocols as set of actions, or oracles, that the attacker can trigger and provide inputs to. Users can define arbitrary protocols, functions and axioms. It models the possible traces of protocols through recursive functions. Without additional axioms, all function symbols are by default interpreted as Probabilistic Polynomial-Time Turing Machine (PPTM), and SQUIRREL has for instance a built-in function symbol modeling the polynomial-time adversary. By design, SQUIRREL's logic hides away the fact that one is manipulating PPTM and probabilities: dedicated axioms have been proven sound w.r.t. to this model, and one can simply rely on those axioms. No built-in axiom restrict the attacker to be weaker than a PPTM, but as in any mechanized prover, users must take care to not declare such axioms, or more generally contradictory or simply false axioms.

In SQUIRREL, we can express and then use arbitrary logical formulas as lemmas, for instance to express some unreachability properties over the protocol, invariants about the states of the protocol, indistinguishability properties, *etc.* Lemmas are then proven in a similar fashion as in, *e.g.*, the Rocq proof assistant [44]: one progresses on the current proof goal, that is the set of current logical hypothesis and conclusion, by applying tactics, where each tactic corresponds to a valid logical rule. SQUIRREL notably allows proofs by induction, *e.g.*, to reason on the possible values of a state, or to show that all possible executions of length $n$ of two protocols are indistinguishable. This is the main proof scheme used in our case-studies.

We note that SQUIRREL provides a set of tactics dedicated to using cryptographic axioms. However, as ideal functionalities generally do not rely on cryptographic functions, our proofs only rely on logical or probabilistic arguments. As such, they could even be argued to hold for stronger adversaries

than PPTM. Though correct w.r.t. to its underlying logic, SQUIRREL cannot for the moment formalize such a statement and one needs to manually inspect which tactics and rules have been used in the proof to argue that the proof holds for stronger adversaries. All our models and proofs are available at [45].

## IV. A DETAILED EXAMPLE: AN ELECTRONIC VOTING FUNCTIONALITY

In this section we illustrate our methodology on a simple example: an ideal functionality for electronic voting. We focus in particular on vote privacy: it should be impossible to link a voter to the vote they cast.

### A. An ideal vote functionality

We first recall an ideal voting system functionality, $\mathcal{F}_{\text{VS}}^{\rho}$, introduced in [12] and depicted in Figure 2. This functionality involves two different types of parties: the set of voters $\mathbf{V}$ and the authority $\mathsf{A}$. $\mathcal{F}_{\text{VS}}^{\rho}$ is parameterized by a result function $\rho$, which takes as input a list of pairs $(V_i, v_i)$ of voters and their votes, and returns the election result, which depends on the election method. This method could be simply the multiset of votes, or a more sophisticated ranking-based method. We also provide $\rho$ with the list of identities, allowing it to take revote policies into account (such as only the last vote counts).

**Remark 1.** We slightly deviate from the presentation in [12] on two aspects. Firstly, we consider only a single authority $\mathsf{A}$, rather than a set of authorities that must all contribute to the tally. This is not a limitation of our framework, but slightly simplifies the presentation of this case study. Secondly, unlike [12], where the revote policy is fixed, we treat it as part of $\rho$. As a consequence, we provide $\rho$ with a list of identity and vote pairs, while the original presentation only provides the list of votes, which directly implies that the result may not depend on the identities, but merely on the votes.

$\mathcal{F}_{\text{VS}}^{\rho}$ collects a voter's votes through the VOTE interface and for each vote leaks the voter's identity to the adversary, but *not* its vote. The votes are however blocked (and will not be counted) until the adversary explicitly authorizes a given identity through the NO-BLOCK interface. Hence, the adversary may block the votes of a particular voter. The TALLY interface allows the authorities to request the result for the votes of all no-blocked voters to be computed using $\rho$, and the result to be published. This functionality is intended to guarantee "by design" perfect ballot secrecy, eligibility and correctness. Specifically, for ballot secrecy, it ensures that *nothing beyond* the voters' identities and the result, *i.e.*, the output of $\rho$, is leaked.

### B. Game-based definition of vote-privacy

Even though it may be "obvious" that $\mathcal{F}_{\text{VS}}^{\rho}$ satisfies vote privacy, we argue that such subtle properties should be verified nevertheless. Moreover, the proof should be straightforward, as ideal functionalities typically do not involve cryptography.

We define a privacy game, inspired by a definition by Benaloh [46], [47]. In this definition, $\rho$ is simply the multiset of votes, with a revote policy that only counts the last vote (even though any other revote policy could be chosen). We denote this particular tally function as $\rho^{\mathsf{ms}}$. As discussed in [20], the definition may be too weak for other tally functions. The privacy game $\mathcal{G}_{\mathsf{Vote-priv}}^{\mathcal{A},\Pi,\beta}$ (Figure 3) depends on a challenge bit $\beta$, and the attacker $\mathcal{A}$'s objective is to distinguish with non-negligible probability, whether $\beta = 0$ or $\beta = 1$, while interacting with a protocol $\Pi$. More formally, we require that

$$\left|\Pr\left[\mathcal{G}_{\mathsf{Vote-priv}}^{\mathcal{A},\Pi,1} = 1\right] - \Pr\left[\mathcal{G}_{\mathsf{Vote-priv}}^{\mathcal{A},\Pi,0} = 1\right]\right|$$

is negligible. In our analysis, $\Pi$ will be instantiated with $\mathcal{F}_{\text{VS}}^{\rho^{\mathsf{ms}}}$.

First, the challenger initializes all machines and the attacker chooses the sets of (dis)honest voters and the authority's identity. The attacker can then interact with the protocol through 3 oracles:

- Guess: at any time the attacker may attempt to guess the challenge bit $\beta$.
- HonestVote: the attacker may instruct an honest voter $V$ to vote. For this, $\mathcal{A}$ provides two votes $v_0$ and $v_1$ and voter $V$ votes for $v_\beta$. Intuitively, the attacker's goal will be to guess whether $v_0$ or $v_1$ was cast. The challenger also maintains two lists $\mathsf{Voted}_0$ and $\mathsf{Voted}_1$ of honestly cast votes, representing the two "possible worlds" depending on $\beta$. Note that the attacker may directly vote on behalf of the dishonest voters.
- HonestTally: At any moment the attacker may request the tally. Crucially, this request is only transmitted if the result of honest votes is the same in both worlds. This avoids the attacker *trivially* winning, by *e.g.*, scheduling a single voter to vote either $v_0$ or $v_1$, resulting into the tally $\{\!\{v_\beta\}\!\}$.

We note that this game directly interfaces with any protocol having the same interface as the ideal functionality. However, interestingly, the security goals are expressed quite differently. While, intuitively, the ideal functionality expresses that nothing more than the result may be learnt, the security game explicitely encodes that the attacker cannot link a voter to their vote, unless this link is trivial (as the votes of honest voters are dictated by $\mathcal{A}$).

**Insecurity of $\mathcal{F}_{\text{VS}}^{\rho}$.** $\mathcal{F}_{\text{VS}}^{\rho^{\mathsf{ms}}}$ does not satisfy vote privacy, as defined by the game. Indeed, the definition does not consider No-Block operations (as all votes are anyway triggered by $\mathcal{A}$). Therefore, the test that $\rho(\mathsf{Voted}_0) = \rho(\mathsf{Voted}_1)$ becomes ineffective if some voters are blocked. For instance, $\mathcal{A}$ may call $\mathsf{HonestVote}(sid, V_1, a, b)$, and $\mathsf{HonestVote}(sid, V_2, b, a)$. In both cases, whether $\beta = 0$ or $\beta = 1$, the resulting tally corresponds to $\{\!\{a, b\}\!\}$. However, if $\mathcal{A}$ writes $(\text{NO-BLOCK}, sid, V_1)$ on $\mathcal{F}_{\text{VS}}^{\rho^{\mathsf{ms}}}$'s backdoor, but not $(\text{NO-BLOCK}, sid, V_2)$, the result directly leaks the challenge bit.

The violation of the security game may at first sight look like a modeling artefact. Indeed, the tally condition should be updated to only compare the result of honest, *non blocked* voters. This is however not possible as the functionality completely hides which voters are blocked or not. Actually, hiding NO-BLOCKs allows the following "broken" protocol to realize

**Functionality $\mathcal{F}_{\text{VS}}^{\rho}(\mathbf{V}, \mathsf{A})$**

- Upon receiving $(\text{VOTE}, sid, v)$ from voter $V \in \mathbf{V}$ and if $v$ is a valid vote, store $(\mathsf{Vote}, sid, v, V)$ and send $(\text{VOTE}, sid, V)$ to $\mathcal{S}$. (Note that the vote $v$ itself is *not* sent to the adversary).
- Upon receiving $(\text{NO-BLOCK}, sid, V)$ from $\mathcal{S}$, store it for future reference.
- Upon receiving $(\text{TALLY}, sid)$ from the administrator $A$, compute the tallying function on all the stored votes $(\mathsf{Vote}, sid, v_i, V_i)$ for which a matching $(\text{NO-BLOCK}, sid, V_i)$ was stored. Calculate $t = \rho((V_1, v_1), \ldots, (V_n, v_n))$ and send $(\text{RESULT}, sid, t)$ to all parties including $\mathcal{S}$ and halt.

Figure 2: Voting ideal functionality [12].

$\mathcal{F}_{\text{VS}}^{\rho,\text{ms}}$. Suppose a protocol that upon receiving $(\text{TALLY}, sid)$ only returns the last vote of a selected voter, say $\mathsf{V}_0$. A formal description of the protocol is depicted in Figure 4. Such a protocol realizes $\mathcal{F}_{\text{VS}}^{\rho,\text{ms}}$ by considering a simulator that sends $(\text{NO-BLOCK}, sid, V_0)$, but no other NO-BLOCK instruction, and otherwise behaves as $\mathcal{A}$. Arguably, this protocol breaks ballot secrecy since $\mathsf{V}_0$'s vote is directly published. In our example, the problematic behavior is of course obvious and easy to detect, but it could be hidden within the specification of a more complex protocol, making it easy to miss.

**Fixed functionality.** We therefore suggest to update $\mathcal{F}_{\text{VS}}^{\rho}$ into $\mathcal{F}_{\text{VS}}^{\rho,\text{fix}}$ by modifying the second item as follows:

- Upon receiving $(\text{NO-BLOCK}, sid, V)$ from $\mathcal{S}$, store it for future reference and send $(\text{NO-BLOCK}, sid, V)$ to $V$.

With this modification, it becomes visible which votes will be counted. Moreover, this modification also allows to take into account notions of individual verifiability where voters may check whether their ballots have been recorded or not. With this modification, we can also update the security game to handle NO-BLOCK instructions: we record any NO-BLOCK actions and check whether $\rho(\mathsf{NBVoted}_0) = \rho(\mathsf{NBVoted}_1)$ before tallying, where $\mathsf{NBVoted}_i$ is the sublist of $\mathsf{Voted}_i$ of unblocked voters. For completeness, we provide in Appendix A the updated version of the privacy game.

## V. CASE-STUDIES

We now consider more complex ideal functionalities, from various applications: anonymous lottery, smart metering, and MPC with identifiable aborts. We show how to challenge their security with game-based security definitions. Applying our approach, we discover that several ideal functionalities actually do not satisfy the properties they mean to capture.

### A. Lotteries in Proof of Stake (PoS)

A key feature of blockchains is their distributed ledger, that anyone trusts. Who can write a new transaction on this board ledger depends on the type of blockchain. In Proof of Stake (PoS) blockchains, at each new round, participants are selected based on the amount of stake (*e.g.*, tokens) they own in the system. More specifically, they are selected via a lottery where their chance of winning depends on their stake. Several protocols [13], [14] have been proposed to ensure the anonymity of participants (who is selected should remain private), as anonymity came for free in Proof of Work blockchains. We examine the two corresponding ideal functionalities and develop corresponding game-based properties that capture anonymity. Our analysis reveals that the ideal functionality presented in [13] does not, in fact, preserve anonymity due to a missing check in the functionality.

*1) Anonymous lottery by [13]:* The anonymous selection functionality $\mathcal{F}_{\text{Anon-Selection}}^{\text{Eligible}}$ of [13] is defined in Figure 5. Intuitively, when a party $P_i$ wants to know whether they are eligible for the next round $\mathsf{tag}$, they query the functionality with $(\text{ELIGIBILITYCHECK}, sid, \mathsf{tag})$. If not already decided, the functionality determines the eligibility at random[2] and returns the result (yes/no) only to $P_i$. Then, parties may create (anonymous) proofs of their eligibility to send a message $msg$ in round $\mathsf{tag}$ by sending $(\text{CREATEPROOF}, sid, \mathsf{tag}, msg)$. To model the fact that the proof can be arbitrary, the adversary can choose any arbitrary message $\psi$ as a proof. Participants may also check proofs by sending $(\text{VERIFY}, sid, \pi, \mathsf{tag}, msg)$ to the functionality.

The identity of the party is never leaked, hence intuitively, this functionality should preserve anonymity.

**Insecurity of $\mathcal{F}_{\text{Anon-Selection}}^{\text{Eligible}}$.** Unfortunately, while $\mathcal{F}_{\text{Anon-Selection}}^{\text{Eligible}}$ does not explicitly leak who is eligible, it lets an attacker deduce it. The anonymity breach attack works as follows:

- At some point, the adversary receives $(\text{PROOF}, \mathsf{tag}, msg)$. It deduces that some (unknown) honest participant $P$ wants to generate a proof of eligibility for a message $msg$ at round $\mathsf{tag}$.
- The adversary produces some (random) proof $\psi$ for $msg$ and sends $(\text{DONE}, \psi, \mathsf{tag}, msg)$ to $\mathcal{F}_{\text{Anon-Selection}}^{\text{Eligible}}$.
- They then request to verify that $msg$ comes from an eligible party but with a false proof $\psi^*$, by sending $(\text{VERIFY}, sid, \psi^*, \mathsf{tag}, msg)$ on behalf of a dishonest party $P'$.
- Since $\psi^*$ is invalid (not stored in the list of valid proofs $\mathcal{L}$), the functionality sends $(\text{VERIFY}, sid, (\psi^*, \mathsf{tag}, msg))$ to the adversary, who needs to produce a witness $w$ for the proof, that should contain an eligible party.
- Here, the adversary tries to guess who was eligible and sends $w = (P_i, \mathsf{tag}, msg)$ as a witness.
  - If $P_i$ was indeed eligible, then the adversary receives $(\text{VERIFIED}, sid, (\psi^*, \mathsf{tag}, msg), 1)$,

[2]For simplicity, [13] considers that all parties have the same stake.

**Game** $\mathcal{G}^{\mathcal{A},\Pi,\beta}_{\mathsf{Vote-priv}}$

– $\mathbf{V} \leftarrow \mathcal{A}$ with $\mathbf{V} = \{\mathsf{V}_1 \ldots, \mathsf{V}_n\}$ (the voters)
– $\mathcal{H}_\mathbf{V} \leftarrow \mathcal{A}$ with $\mathcal{H}_\mathbf{V} \subseteq \mathbf{V}$ (the honest voters)
– $\mathsf{A} \leftarrow \mathcal{A}$ (the authority)
– $\mathcal{C}$ initializes $\Pi(\mathbf{V}, \mathsf{A})$ with back-doors linked with $\mathcal{A}$
– $\mathcal{C}$ initializes empty lists $\mathsf{Voted}_0$, $\mathsf{Voted}_1$ which are part of the Oracles shared state
– $\mathcal{C}$ initializes the flag $\mathsf{tallied}$ to $\perp$ which is part of the Oracles shared state

Upon receiving $\mathsf{Guess}(sid, \beta^*)$ from $\mathcal{A}$
// $\mathcal{A}$'s guess received on $\mathcal{C}$'s backdoor tape
Output $\beta^*$

Upon receiving $\mathsf{HonestTally}(sid)$ from $\mathcal{A}$
If $\rho^{\mathsf{ms}}(\mathsf{Voted}_0) = \rho^{\mathsf{ms}}(\mathsf{Voted}_1)$
Send $(\textsc{Tally}, sid)$ to $\Pi$ on behalf of $\mathsf{A}$
$\mathsf{tallied} = \top$

Upon receiving $\mathsf{HonestVote}(sid, V, v_0, v_1)$ from $\mathcal{A}$
If $V \in \mathcal{H}_\mathbf{V}$ and $\neg\mathsf{tallied}$
Send $(\textsc{Vote}, sid, v_\beta)$ to $\Pi$ on behalf of $V$
$\mathsf{Voted}_0 \leftarrow \mathsf{Voted}_0@[(V, v_0)]$
$\mathsf{Voted}_1 \leftarrow \mathsf{Voted}_1@[(V, v_1)]$

Figure 3: Game $\mathcal{G}^{\mathcal{A},\Pi,\beta}_{\mathsf{Vote-priv}}$ for vote privacy.

**Protocol** $\Pi_{\mathrm{BV}}(\mathbf{V}, \mathsf{A})$

– Upon receiving $(\textsc{Vote}, sid, v)$ from voter $V \in \mathbf{V}$ and if $v$ is valid, store $(\mathsf{Vote}, sid, v, V)$ and send $(\textsc{Vote}, sid, V)$ to $\mathcal{A}$.
– Upon receiving $(\textsc{Tally}, sid)$ from the administrator $A$, let $t = \{\!\{v\}\!\}$ where $(\mathsf{Vote}, sid, v, \mathsf{V}_0)$ is the last vote stored for $\mathsf{V}_0$ (or $t = \emptyset$ if $\mathsf{V}_0$ did not vote). Send $(\textsc{Result}, sid, t)$ to all parties including $\mathcal{A}$ and halt.

Figure 4: Voting protocol violating ballot secrecy of $V_0$.

**Functionality** $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}(\mathcal{P})$

The ideal functionality is parameterized by an $\mathsf{Eligible}$ predicate and maintains the following elements: (1) A global set of registered parties $\mathcal{P} = ((\mathsf{P}_1, \mathsf{stake}_1), \ldots, (\mathsf{P}_n, \mathsf{stake}_n))$. (2) A table $T$, which has one row per party and a column for each $\mathsf{tag} \in \mathbb{N}$ given by parties when checking eligibility. The table stores the eligibility information of each party in each tag. (3) A list $\mathcal{L}$, to store a proof $\pi$ corresponding to a message $msg$ in some tag.

– Upon receiving $(\textsc{EligibilityCheck}, sid, \mathsf{tag})$ from $P$
  1) If $(P, s) \in \mathcal{P}$ and $T(P, \mathsf{tag})$ is undefined, sample $r \in \{0,1\}^l$ run $\mathsf{Eligible}(r, s, \mathsf{tag})$ to get $b \in \{0,1\}$. Set $T(P, \mathsf{tag}) = b$
  2) Send $(\textsc{EligibilityCheck}, T(P, \mathsf{tag}))$ to $P$
– Upon receiving $(\textsc{CreateProof}, sid, \mathsf{tag}, msg)$ from $P$
  1) If $T(P, \mathsf{tag}) = 1$, send $(\textsc{Proof}, \mathsf{tag}, msg)$ to $\mathcal{A}$. Else, send $(\textsc{Declined}, \mathsf{tag}, msg)$ to $P$.
  2) Upon receiving $(\textsc{Done}, \pi, \mathsf{tag}, msg)$ from $\mathcal{A}$. Record $(\pi, \mathsf{tag}, msg)$ in $\mathcal{L}$. Send $(\textsc{Proof}, \pi, \mathsf{tag}, msg)$ to $P$.
– Upon receiving $(\textsc{Verify}, sid, \pi, \mathsf{tag}, msg)$ from $P$
  1) If $(\pi, \mathsf{tag}, msg) \in \mathcal{L}$, send $(\textsc{Verified}, sid, (\pi, tag, msg), 1)$ to $P$.
  2) If $(\pi, \mathsf{tag}, msg) \notin \mathcal{L}$, send $(\textsc{Verify}, sid, (\pi, \mathsf{tag}, msg))$ to $\mathcal{A}$ and wait for a witness $w$ from the adversary $\mathcal{A}$. Check if $w$ is valid as follows:
     – Parse $w = (P', \mathsf{tag}, msg)$ and check that $T(P', \mathsf{tag}) = 1$ and $P'$ is dishonest.
     – If yes, store $(\pi, \mathsf{tag}, msg)$ in the list $\mathcal{L}$ and send $(\textsc{Verified}, sid, (\pi, \mathsf{tag}, msg), 1)$ to $P$.

     If either of these checks are false output $(\textsc{Verified}, sid, (\pi, \mathsf{tag}, msg), 0)$ to $P$.

Figure 5: Ideal functionality $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$ as written in [13]. Proposed fix highlighted, yielding $\mathcal{F}^{\mathsf{Eligible,fix}}_{\mathsf{Anon-Selection}}$.

– Else, $P_i$ was not eligible, and the adversary receives $(\textsc{Verified}, sid, (\psi^*, \mathsf{tag}, msg), 0)$.

- The adversary can hence deduce whether $P_i$ was eligible or not, which breaks anonymity.

One may wonder why $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$ asks the adversary to produce a witness when the proof $\psi^*$ is invalid (not already

stored in $\mathcal{L}$) in the first place? This is to account for the fact that a dishonest party may forge valid proofs when they are indeed eligible. However, only *dishonest* parties should be able to forge such proofs. Honest parties must use the $(\text{PROOF}, \mathsf{tag}, msg)$ query.

Hence it is easy to fix the functionality: $\mathcal{F}^{\mathsf{Eligible,fix}}_{\mathsf{Anon-Selection}}$ checks that the party $P_i$ in the witness $w$ provided by the adversary is indeed dishonest. $\mathcal{F}^{\mathsf{Eligible,fix}}_{\mathsf{Anon-Selection}}$ is formally defined in Figure 5, with the fixes highlighted.

**Game-based definition of anonymity.** This attack would have been caught if the anonymity property of $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$ had been formally studied. For this, we define the anonymity game for the lottery in Figure 6, following a standard indistinguishability game. Intuitively, the attacker tries to distinguish whether they are interacting with $P_0$ or $P_1$. More specifically, the adversary may schedule honest participants $P_i, P_j$ to create proofs through the oracle $\mathsf{HonestCreateProof}(sid, \mathsf{tag}, msg, P_i, P_j)$. In that case, they actually schedule $P_i$ in the world that corresponds to $\beta = 0$ and $P_j$ in the world that corresponds to $\beta = 1$. While the adversary has scheduled the participants themself, they should not be able to tell which world they see. Moreover, the adversary may schedule participants to run the eligibility protocol. When they interact with the real functionality ($\beta' = 0$), eligibility is determined by the functionality, while when $\beta' = 1$, they interact with an ideal setting where eligibility purely follows the $\mathsf{Eligible}$ parameter predicate initialized with a random $r$. This models that the attacker may learn nothing more than what is leaked by $\mathsf{Eligible}$. Note that even if the same distribution is used when $\beta' = 0$ or $\beta' = 1$, participants are likely to be different at each call. Also note that we let the adversary choose $\beta'$ when calling $\mathsf{HonestEligibilityCheck}$. An alternative definition could define both $\mathsf{Eli}_0$ and $\mathsf{Eli}_1$ at each call. The choice to let the adversary provide $\beta'$ yields however a stronger definition that we were able to prove on the fixed functionality.

We can show that $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$ does not satisfy $\mathcal{G}_{\mathsf{Lottery}}$ (details in Appendix B) while $\mathcal{F}^{\mathsf{Eligible,fix}}_{\mathsf{Anon-Selection}}$ does, which is proved using SQUIRREL, as explained in Section VI.

*2) Anonymous lottery by [14]:* Another ideal functionality $\mathcal{F}^{\mathsf{LE}}_{\mathsf{Lottery}}$ is proposed in [14] to model the security of an anonymous lottery. As in $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$, the functionality $\mathcal{F}^{\mathsf{LE}}_{\mathsf{Lottery}}$ determines who is eligible at random, depending on the stake of the parties. The rest of the functionality is a bit more abstract than in $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$: whenever a party $P_i$ wishes to send a message, $\mathcal{F}^{\mathsf{LE}}_{\mathsf{Lottery}}$ checks that $P_i$ is eligible and if so, broadcasts the message to all parties. Parties may read their incoming messages and the adversary may selectively send a message to a particular party, in the name of a dishonest stakeholder pid, if pid is eligible.

Since $\mathcal{F}^{\mathsf{LE}}_{\mathsf{Lottery}}$ and $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$ do not have exactly the same interface, we need to slightly adapt our game $\mathcal{G}^{\mathcal{A},\Pi,\beta}_{\mathsf{Lottery}}$ to $\mathcal{F}^{\mathsf{LE}}_{\mathsf{Lottery}}$. Both $\mathcal{F}^{\mathsf{LE}}_{\mathsf{Lottery}}$ and the adapted game are formally provided in Appendix B. We prove using SQUIRREL that $\mathcal{F}^{\mathsf{LE}}_{\mathsf{Lottery}}$ does preserve anonymity by showing that it satisfies the (adapted) game $\mathcal{G}_{\mathsf{Lottery}}$, as explained in Section VI.

### B. SMPC with identifiable abort

The goal of Secure Multi Party Computation (SMPC) is to evaluate a function $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ on a vector of $n$ private inputs to a vector of $n$ outputs. Specifically, each party $P_j$ provides a secret input $i_j$ and receives the corresponding output $o_j$, where $f(i_1, \ldots, i_n) = (o_1, \ldots, o_n)$. A crucial property of SMPC is *privacy*: a participant $P_j$ must not learn anything about the inputs and outputs of any other party beyong what is revealed by $o_j$.

Another desirable property is *fairness*: a corrupted party receives its output if and only if all honest parties also do. Unfortunately, without an honest majority, fairness cannot be achieved [48]. Therefore, protocols have been designed to achieve so-called *security with abort*: either all honest parties obtain the result, or they all learn that the protocol aborted.

We describe in Figure 7 a classical ideal functionality for security with abort, denoted $\mathcal{F}^f_{\text{SMPC}}(\mathcal{P})$, described, *e.g.*, in [49, App. A.2], but using a notational style similar to the other examples in this paper. The communication model assumes a *synchronous network with rushing adversary*: the execution proceeds in rounds, and each output depends only on inputs received in previous rounds; an exception is made for the adversary, whose output may depend on inputs from the *current* round, a capability referred to as rushing. To model such a synchronous network we explicitly maintain the round number $\mathsf{r}$ in the functionality. Note that when all inputs have been received the output of corrupted parties is immediately leaked, which corresponds to the absence of fairness.

*Identifiable abort:* A useful variant of *security with abort* adds accountability for misbehaving parties by including the notion of *identifiability* [15]. Exposing the identity of misbehaving parties is expected to disincentive denial-of-service attacks, which occur when such a party forces an abort. In [15], security with abort is defined as

> *Either the protocol succeeds, in which case every party receives its output, or the protocol aborts, in which case all honest parties learn that the protocol aborted.*

and identifiable abort

> *leverages the effect of an abort by forcing, upon abort, at least one malicious party to reveal its identity.*

Ishai *et al.* [15] propose a compiler that transforms any UC functionality $\mathcal{F}$, into the functionality $[\mathcal{F}]^{\mathsf{ID}}_{\perp}$ which behaves identically to $\mathcal{F}$, with one exception: at any time the adversary may send a special command $(\text{ABORT}, P_j)$, where $j \in \mathcal{I}$ is the identity of one of the corrupted parties. This command causes $[\mathcal{F}]^{\mathsf{ID}}_{\perp}$ to output $(\text{ABORT}, P_j)$ to all parties. We do not describe here the compiler itself, but apply it to $\mathcal{F}^f_{\text{SMPC}}(\mathcal{P})$ resulting in $[\mathcal{F}^f_{\text{SMPC}}]^{\mathsf{ID}}_{\perp}(\mathcal{P})$ which is detailed in Figure 8, and also described in [51]. Intuitively, when a party $P_j$ aborts, $[\mathcal{F}]^{\mathsf{ID}}_{\perp}$ outputs $(\text{ABORT}, P_j)$ to each party instead of ABORT.

**Insecurity of $[\mathcal{F}^f_{\text{SMPC}}]^{\mathsf{ID}}_{\perp}(\mathcal{P})$ and game based definition.** As we will now show, it is, however, possible for an attacker

**Game** $\mathcal{G}^{\mathcal{A},\Pi,\beta}_{\mathsf{Lottery}}$

- $\mathcal{P} \leftarrow \mathcal{A}$ with $\mathcal{P} = \{(\mathsf{P}_1, \mathsf{stake}_1), \ldots, (\mathsf{P}_n, \mathsf{stake}_n)\}$ (the parties)
- $\mathcal{H} \leftarrow \mathcal{A}$ with $\mathcal{H} \subseteq \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ (the honest parties)
- $\mathcal{C}$ initialises $\Pi(\mathcal{P})$ with back-doors linked with $\mathcal{A}$
- $\mathcal{C}$ initialises empty tables $\mathsf{Eli}_0$ and $\mathsf{Eli}_1$ which are part of the Oracles shared state

Upon receiving $\mathsf{Guess}(sid, \beta^*)$ from $\mathcal{A}$
// $\mathcal{A}$'s guess received on $\mathcal{C}$'s backdoor tape
Output $\beta^*$

Upon receiving $\mathsf{HonestEligibilityCheck}(sid, \mathsf{tag}, P, \beta')$ from $\mathcal{A}$
If $P \in \mathcal{H}$
If $\beta' = 0$
Send $(\textsc{EligibilityCheck}, sid, \mathsf{tag})$ to $\Pi$ on behalf of $P$
Wait for $(\textsc{EligibilityCheck}, b)$
If $\mathsf{Eli}_{\beta'}(P, \mathsf{tag})$ is undefined
$\mathsf{Eli}_{\beta'}(P, \mathsf{tag}) \leftarrow b$
Else // $\beta' = 1$
If $\mathsf{Eli}_{\beta'}(P, \mathsf{tag})$ is undefined
Sample $r \in \{0,1\}^l$
Run $\mathsf{Eligible}(r, \mathsf{stake}_i, \mathsf{tag})$ to get $b \in \{0,1\}$
$\mathsf{Eli}_{\beta'}(P, \mathsf{tag}) \leftarrow b$

Upon receiving $\mathsf{HonestCreateProof}(sid, \mathsf{tag}, msg, P_0, P_1)$ from $\mathcal{A}$
If $P_0, P_1 \in \mathcal{H}$ and $\mathsf{Eli}_0(P_0, \mathsf{tag})$ and $\mathsf{Eli}_1(P_1, \mathsf{tag})$ are defined and $\mathsf{Eli}_0(P_0, \mathsf{tag}) = \mathsf{Eli}_1(P_1, \mathsf{tag})$
Send $(\textsc{CreateProof}, sid, \mathsf{tag}, msg)$ to $\Pi$ on behalf of $P_\beta$
Wait for answer $(\textsc{Declined}, \mathsf{tag}, msg)$ or $(\textsc{Proof}, \pi, \mathsf{tag}, msg)$

Upon receiving $\mathsf{HonestVerify}(sid, \pi, \mathsf{tag}, msg, P)$ from $\mathcal{A}$
If $P \in \mathcal{H}$
Send $(\textsc{Verify}, sid, \pi, \mathsf{tag}, msg)$ to $\Pi$ on behalf of $P$
Wait for the answer $(\textsc{Verified}, sid, (\pi, \mathsf{tag}, msg), 0)$

Figure 6: Game $\mathcal{G}^{\mathcal{A},\Pi,\beta}_{\mathsf{Lottery}}$ for anonymous lottery.

**Functionality** $\mathcal{F}^f_{\text{SMPC}}(\mathcal{P})$

Let $n = |\mathcal{P}|$, and $\mathcal{D}$ the domain of $f$. Initialize $\beta_j, i_j, o_j \leftarrow \perp$ for all $j \in [n]$. Initialize $\mathsf{r} \leftarrow 0$. Initialize $\alpha \leftarrow \textsc{Init}$. $\mathcal{S}$ corrupts parties $\mathcal{I} \subseteq \mathcal{P}$.

- Upon receiving $(\textsc{Input}, sid, x_j)$ from party $P_j \in \mathcal{P} \setminus \mathcal{I}$ or from $\mathcal{S}$ on behalf of $P_j \in \mathcal{I}$, if $\beta_j = \perp$ then set $\beta_j \leftarrow \top$, set $i_j \leftarrow x_j$, and set $\mathsf{r} \leftarrow 1$.
  If $(\bigwedge_{j\in\mathcal{P}} \beta_j)$ and $(\bigwedge_{j\in\mathcal{P}} (i_j \in \mathcal{D}))$, then set $\mathsf{r} \leftarrow 2$, and set $(o_1, \ldots, o_n) \leftarrow f(i_1, \ldots, i_n)$.
  If $(\bigwedge_{j\in\mathcal{P}} \beta_j)$ and $\neg(\bigwedge_{j\in\mathcal{P}} (i_j \in \mathcal{D}))$, then set $\mathsf{r} \leftarrow 2$, and set $(o_1, \ldots, o_n) \leftarrow (\textsc{InvalidInput}, \ldots, \textsc{InvalidInput})$.
  If $(\bigwedge_{j\in\mathcal{P}} \beta_j)$, then send $((\textsc{Output}, sid, \mathcal{O}_\mathcal{I}))$ to $\mathcal{S}$ where $\mathcal{O}_\mathcal{I} = \{(P_j, o_j) \mid P_j \in \mathcal{I}\}$.
- Upon receiving $(\textsc{Allow}, sid)$ from $\mathcal{S}$ on behalf of $P_j \in \mathcal{I}$, if $(\bigwedge_{j\in\mathcal{P}} \beta_j) = \top$ and $\alpha = \textsc{init}$, then set $\alpha \leftarrow \textsc{Allow}$.
- Upon receiving $(\textsc{Abort}, sid)$ from $\mathcal{S}$ on behalf of $P_j \in \mathcal{I}$, if $(\bigwedge_{j\in\mathcal{P}} \beta_j) = \top$ and $\alpha = \textsc{init}$, then set $\alpha \leftarrow \textsc{Abort}$, and set $(o_1, \ldots, o_n) \leftarrow (\textsc{Abort}, \ldots, \textsc{Abort})$.
- Upon receiving $(\textsc{Output}, sid)$ from party $P_j \in \mathcal{P} \setminus \mathcal{I}$, if $\mathsf{r} = 2$ and $\alpha = \textsc{Init}$, then return $(\textsc{Output}, sid, \textsc{Waiting})$, else return $(\textsc{Output}, sid, o_j)$.
- Upon receiving $(\textsc{Round}, sid)$ from party $P_j \in \mathcal{P} \setminus \mathcal{I}$ or from $\mathcal{S}$ on behalf of $P_j \in \mathcal{I}$, return $(\textsc{Round}, sid, \mathsf{r})$.

Figure 7: UC-compliant ideal SMPC functionality corresponding to secure computation with abort and no fairness in the taxonomy of [50].

to learn the outputs of dishonest participants while honest participants remain unable to identify one of the dishonest participants. This occurs because the adversary/simulator may obtain the result of the computation without explicitly aborting nor triggering an allow message, to let the protocol proceed. More formally, consider three parties, with the last one being dishonest: $\mathcal{P} = \{P_1, P_2, P_3\}$ and $\mathcal{I} = \{P_3\}$.

- $P_1$ and $P_2$ send $(\textsc{Input}, sid, v_1$ (resp. $v_2))$ to the functionality, where $v_1$ (resp. $v_2$) $\xleftarrow{\$} \mathcal{D}$.
- $\mathcal{A}$ sends $(\textsc{Input}, sid, v_3)$ for some $v_3 \xleftarrow{\$} \mathcal{D}$ and waits for $((\textsc{Output}, sid, \{o_3\}))$, then halts.

Note that when $\mathcal{A}$ halts, the control goes back to the environment, and $\mathcal{A}$ will not become active anymore. Hence, neither ALLOW nor ABORT will be sent to the functionality on behalf of a dishonest participant. It is now impossible for an honest participant to be informed of the (implicit) abort and identify one of the corrupted participants. Indeed, if an honest participant submits $(\textsc{Output}, sid)$, the functionality will return $(\textsc{Output}, sid, \textsc{Waiting})$ leaving them in this undesirable state indefinitely. This is reminiscent of problems in optimistic fair exchange problems, where the ability to always reach a final state is known as timeliness, see *e.g.*, [52]. We argue that this violates identifiable abort, as the protocol neither succeeds, nor do honest parties learn the identity of

**Functionality** $[\mathcal{F}^{f}_{\text{SMPC}}]^{\mathsf{ID}}_{\perp}(\mathcal{P})$

Let $n = |\mathcal{P}|$, and $\mathcal{D}$ the domain of $f$. Initialize $\beta_j, i_j, o_j \leftarrow \perp$ for all $j \in [n]$. Initialise $\mathsf{r} \leftarrow 0$. Initialize $\alpha \leftarrow$ INIT. $\mathcal{S}$ corrupts parties $\mathcal{I} \subseteq \mathcal{P}$.

- Upon receiving $(\text{INPUT}, sid, x_j)$ from party $P_j \in \mathcal{P} \setminus \mathcal{I}$ or from $\mathcal{S}$ on behalf of $P_j \in \mathcal{I}$, if $\beta_j = \perp$ then set $\beta_j \leftarrow \top$, set $i_j \leftarrow x_j$, and set $\mathsf{r} \leftarrow 1$.
  If $(\bigwedge_{j\in\mathcal{P}} \beta_j)$ and $(\bigwedge_{j\in\mathcal{P}} (i_j \in \mathcal{D}))$, then set $\mathsf{r} \leftarrow 2$, and set $(o_1, \ldots, o_n) \leftarrow f(i_1, \ldots, i_n)$.
  If $(\bigwedge_{j\in\mathcal{P}} \beta_j)$ and $\neg(\bigwedge_{j\in\mathcal{P}} (i_j \in \mathcal{D}))$, then set $\mathsf{r} \leftarrow 2$, and set $(o_1, \ldots, o_n) \leftarrow (\text{INVALIDINPUT}, \ldots, \text{INVALIDINPUT})$.
  If $(\bigwedge_{j\in\mathcal{P}} \beta_j) = \top$, then send $((\text{OUTPUT}, sid, \mathcal{O}_{\mathcal{I}}))$ to $\mathcal{S}$ where $\mathcal{O}_{\mathcal{I}} = \{(P_j, o_j) \mid P_j \in \mathcal{I}\}$.
- Upon receiving $(\text{ALLOW}, sid)$ from $\mathcal{S}$ on behalf of $P_j \in \mathcal{I}$, if $(\bigwedge_{j\in\mathcal{P}} \beta_j) = \top$ and $\alpha = \text{INIT}$, then set $\alpha \leftarrow \text{ALLOW}$.
- Upon receiving $(\text{ABORT}, sid, P_j)$ from $\mathcal{S}$ on behalf of $P_j \in \mathcal{I}$ where $P_j \in \mathcal{I}$, if $(\bigwedge_{j\in\mathcal{P}} \beta_j) = \top$ and $\alpha = \text{INIT}$, then set $\alpha \leftarrow \text{ABORT}$, and set $(o_1, \ldots, o_n) \leftarrow ((\text{ABORT}, P_j), \ldots, (\text{ABORT}, P_j))$.
- Upon receiving $(\text{OUTPUT}, sid)$ from party $P_j \in \mathcal{P} \setminus \mathcal{I}$, if $\mathsf{r} = 2$ and $\alpha = \text{INIT}$, then set $\alpha \leftarrow \text{ALLOW}$.
  Return $(\text{OUTPUT}, sid, o_j)$.
- // Upon receiving $(\text{OUTPUT}, sid)$ from party $P_j \in \mathcal{P} \setminus \mathcal{I}$, if $\mathsf{r} = 2$ and $\alpha = \text{INIT}$, then return $(\text{OUTPUT}, sid, \text{WAITING})$, else return $(\text{OUTPUT}, sid, o_j)$.
- Upon receiving $(\text{ROUND}, sid)$ from party $P_j \in \mathcal{P} \setminus \mathcal{I}$ or from $\mathcal{S}$ on behalf of $P_j \in \mathcal{I}$, return $(\text{ROUND}, sid, \mathsf{r})$.

Figure 8: Ideal SMPC functionality with identifiable abort [15]. The fixed functionality ensures liveness, *i.e.* when round $\mathsf{r} = 2$ is reached honest parties get the output of the computation, unless the adversary has aborted on behalf of one of the corrupted parties.

a malicious party. A very simple fix is proposed in Figure 8: if the attacker has not chosen between ALLOW or ABORT, possibly because they halted, and an honest participant requests the result we set the internal state $\alpha$ to ALLOW and return the result. One may also note that the fix has similarities with the protocol proposed by Brandt *et al.* [53] which also prevents honest parties from being left "hanging".

This problem could have been spotted by verifying the game we define in Section V-B, which can be proven on the fixed version. $\mathcal{A}$ wins the game if the challenger outputs 1. $\mathcal{A}$ may interact with the challenger to schedule honest participants to provide their input, request their output, and query the round. $\mathcal{A}$ may also act directly on behalf of the dishonest parties by writing on the functionality's backdoor tape. At any time, $\mathcal{A}$ can challenge honest participant $P$. The challenger will then check the round number $\mathsf{r}$ and request the output for $P$. The attacker wins in 3 cases: *(i)* we have reached the final round ($\mathsf{r} = 2$), but the honest participant is still in state WAITING; *(ii)* the protocol has been aborted, but an *honest* participant is identified as misbehaving; *(iii)* the protocol has been aborted, but not all honest participants agree on the same misbehaving participant. This game allows us to capture the problem we identified earlier on $[\mathcal{F}^{f}_{\text{SMPC}}]^{\mathsf{ID}}_{\perp}(\mathcal{P})$, and we prove using SQUIRREL that the attacker cannot win this game.

Taking a step back, we further note that any protocol $\Pi$ UC-realizing a functionality $\mathcal{F}$—even one offering no form of identifiable abort—will also UC-realize $[\mathcal{F}]^{\mathsf{ID}}_{\perp}$. This occurs because the ideal functionality $[\mathcal{F}]^{\mathsf{ID}}_{\perp}$ enforces no observable difference unless the simulator explicitly issues the $(\text{ABORT}, sid, P_j)$ command, which it is never compelled to do. Therefore, the additional structure intended for identifiable abort becomes vacuous within the UC framework as defined, as the same protocol and simulator pair suffice to realize both $\mathcal{F}$ and $[\mathcal{F}]^{\mathsf{ID}}_{\perp}$.

### C. Privacy-preserving smart metering

Smart meters allow fine-grained tariff policies. A user might have meters from multiple companies (*e.g.*, gaz and electricity) and could be charged differently based on consumption time or a consumption threshold. However, revealing a user's exact consumption for each of their meters poses a privacy threat. Therefore, privacy-preserving billing protocols have been proposed to facilitate fair billing without disclosing private information to the companies. To analyse the security of such protocols, [3] introduces an ideal functionality $\mathcal{F}_{\mathsf{BIL}}$. This complex functionality reflects the various operations of these protocols.

Privacy-preserving billing protocols should guarantee that a dishonest user, controlling some of their meters, cannot pay less than the price charged by their honest meters. Conversely, when all the meters are honest, an honest user should not be charged more than the price corresponding to their actual consumption. We formalize this property as a game-based definition $\mathcal{G}_{\mathsf{Billing}}$. When applying $\mathcal{G}_{\mathsf{Billing}}$ to $\mathcal{F}_{\mathsf{BIL}}$, we observe that $\mathcal{F}_{\mathsf{BIL}}$ does not actually satisfy the game. Indeed, a dishonest user can bypass their honest meters and entirely control the charged price, thereby avoiding payment for their actual consumption. We believe that it is an oversight by the authors, likely due to the complexity of $\mathcal{F}_{\mathsf{BIL}}$. Indeed, as soon as one meter is dishonest, the functionality forgets to compute the price charged by the honest meters. This issue can be easily fixed; we demonstrate that the fixed version $\mathcal{F}^{\mathsf{fix}}_{\mathsf{BIL}}$ satisfies $\mathcal{G}_{\mathsf{Billing}}$, using SQUIRREL. All the details are provided in Appendix C.

**Game** $\mathcal{G}^{\mathcal{A},\Pi}_{\mathsf{SMPC}}$

– $\mathcal{P} \leftarrow \mathcal{A}$ with $\mathcal{P} = \{P_1 \ldots, P_n\}$ (the parties)
– $f \leftarrow \mathcal{A}$ with $\mathcal{D}$ domain of $f$
– $\mathcal{I} \leftarrow \mathcal{A}$ with $\mathcal{I} \subseteq \mathcal{P}$ (the dishonest parties)
– $\mathcal{C}$ initialises $\Pi$ with back-doors linked with $\mathcal{A}$
– $\mathcal{C}$ initialises $\beta \leftarrow 0$

Upon receiving $(\textsc{Challenge}, sid, P)$ from $\mathcal{A}$
  If $P \in \mathcal{P} \setminus \mathcal{I}$ // $P$ is honest
    Send $(\textsc{Round}, sid)$ to $\Pi$ on behalf of $P$
    Wait for $(\textsc{Round}, sid, \mathsf{r})$ from $\Pi$ on behalf of $P$
    Send $(\textsc{Output}, sid)$ to $\Pi$ on behalf of $P$
    Wait for $(\textsc{Output}, sid, out)$ from $\Pi$ on behalf of $P$
    If $\mathsf{r} = 2$ and $out = \textsc{Waiting}$
      Set $\beta \leftarrow 1$
    If $\mathsf{r} = 2$ and $out = (\textsc{Abort}, P^*)$
      If $P^* \notin \mathcal{I}$
        Set $\beta \leftarrow 1$
      Else
        For all $P' \in \mathcal{P} \setminus \mathcal{I}$
          Send $(\textsc{Output}, sid)$ to $\Pi$ on behalf of $P'$
          Wait for $(\textsc{Output}, sid, out')$ from $\Pi$ on behalf of $P'$
          If $out' \neq out$ // $out = (\textsc{Abort}, P^*)$
            Set $\beta \leftarrow 1$
    Output $\beta$

Upon receiving $(\textsc{HonestInput}, sid, P, x)$ from $\mathcal{A}$
  If $P \in \mathcal{P} \setminus \mathcal{I}$ // $P$ is honest
    Send $(\textsc{Input}, sid, x)$ to $\Pi$ on behalf of $P$

Upon receiving $(\textsc{HonestOutput}, sid, P)$ from $\mathcal{A}$
  If $P \in \mathcal{P} \setminus \mathcal{I}$ // $P$ is honest
    Send $(\textsc{Output}, sid)$ to $\Pi$ on behalf of $P$

Upon receiving $(\textsc{HonestRound}, sid, P)$ from $\mathcal{A}$
  If $P \in \mathcal{P} \setminus \mathcal{I}$ // $P$ is honest
    Send $(\textsc{Round}, sid)$ to $\Pi$ on behalf of $P$

Figure 9: Security game for SMPC with identifiable abort.

## VI. Proving game-based properties on ideal functionalities

We now recall the constraints on UC-compatible games, formalize how to derive from a UC-compatible game one which follows the usual execution model of cryptographic games, called here fully-explicit game. Finally, we showcase our results obtained with SQUIRREL. We summarize our case studies in Table I. We emphasize that our approach is generally tool agnostic and not specific to SQUIRREL; other tools and techniques could be used to prove that ideal functionalities are secure w.r.t. their game-based properties.

### A. Building fully-explicit game

In the previous sections, we defined security games that can be used to prove that a functionality, seen as a protocol, does satisfy a given security property. We recall that these games, that we call UC-compatible games, must be of the form specified in Section III-B for security to transfer from the ideal functionality to any protocol that realizes it. Notably, a UC-compatible game interacts at the level of the UC environment: it can send and receive messages from honest parties and it can receive arbitrary messages from the attacker. More formally, during the execution of a UC-compatible game $\mathcal{C}$, $\mathcal{C}$ interacts both with $\Pi$ and the attacker $\mathcal{A}$, while $\mathcal{A}$ and $\Pi$ can also interact through backdoors invisible to the game. We must ensure that $\mathcal{C}$ only communicates with honest parties. Therefore, we conditions these games (see, *e.g.*, the condition "if $V \in \mathcal{H}_{\mathbf{V}}$" in Figure 3).

**Classical execution model.** The classical execution model of cryptographic games does not account for hidden backdoor communications between the attacker and the functionality $\Pi$: such games must make explicit all oracles the attacker can interact with. Therefore, we apply a straightforward transformation to a game $\mathcal{C}$ prior to the security proof, which can be either a classical pen-and-paper game-based proof or a computer-assisted proof relying on tools from the literature. Given a UC-compatible game $\mathcal{C}$ and a functionality $\mathcal{F}$, we define the corresponding $\mathcal{C}^{\mathsf{explicit}}$ as follows:

- $\mathcal{C}^{\mathsf{explicit}}$ contains all the oracles from $\mathcal{C}$, where the code of $\Pi$ has been inlined. Note that, as a result of this inlining, a single oracle may be transformed into a set of oracles that the attacker can sequentially interact with.
- In addition, for each command

  Upon receiving $\mathsf{X}(params)$ from $\mathcal{A}$ do $Y$

  in $\mathcal{F}$, we add the oracle

  Upon receiving $\mathsf{CorruptedX}(params)$ from $\mathcal{A}$
    If $P \notin \mathcal{H}$ do $Y$

  where $\mathcal{H}$ is the set of honest parties. This oracle gives explicit access to the backdoors of $\Pi$.

We acknowledge that the above description is not completely formal. However, ideal functionalities in the literature

are written in informal dialects, usually hybrid forms of pseudocode and natural language, which makes the point of fully specifying the transformation moot. It is however very simple and basically consists of inlining. Also note that formally, all oracles in $\mathcal{C}^{\mathsf{explicit}}$ must return a result to the attacker at the end of their execution. For ease of notation we do not always explicitly return a value, but adopt the convention that, if not explicit, oracles implicitly return an empty value.

**E-voting example.** In Section IV-B, we introduced the security game $\mathcal{G}^{\mathcal{A},\Pi,\beta}_{\mathsf{Vote-priv}}$ to prove the security of the (updated) ideal voting functionality $\mathcal{F}^{\rho^{\mathsf{ms}}}_{\mathrm{VS}}$. To produce the corresponding explicit game, we only have to inline the ideal voting functionality in the game, and add the oracle $\mathsf{CorruptedVote}(sid, V, v)$ that directly stores the vote $v$ and models the backdoor access to $\Pi$. The resulting explicit game is given in Appendix A.

### B. Using SQUIRREL

For all the case-studies presented in Sections IV and V, we used SQUIRREL to verify that the proposed game based properties hold on the repaired functionalities. SQUIRREL's input language for protocols is a dialect of the applied pi-calculus, a specification language for parallel processes with explicit input and output commands that allows interaction with the network attacker. We can write fully-explicit games by translating each oracle

Upon receiving $\mathsf{X}(params)$ from $\mathcal{A}$ do $Y$; return $Z$

as a process of the form "$\mathsf{in}(params); Y; \mathsf{out}(Z)$" and composing them in parallel. Moreover, SQUIRREL supports state, abstract functions defined through an axiomatization, as well as specification of indistinguishability and reachability properties. As a result, most of our examples were straightforward to model. Still, we carried out modeling simplifications, either to make the models more readable, ease the proof, or overcome some limitations of SQUIRREL:

- Oracles for honest and dishonest participants were sometimes merged together, to avoid oracles with empty else branches and code duplication.
- When using some state $\mathsf{st}\ i$ to model the state of an agent $P_i$, SQUIRREL does not directly allow updating all the corresponding states at once in an atomic step. In such a case, we spawn an unbounded number of oracles allowing the attacker to update for each $i$ the corresponding $\mathsf{st}\ i$. The security of this game trivially implies the security of the game where all updates happen at once.
- Some functions are "under-axiomatized", which simplifies the model and yields a more general proof. For instance, in the voting case, we do not detail the implementation of $\rho^{\mathsf{ms}}$, but consider an arbitrary function $\rho$ that satisfies a set of basic axioms. Hence, our proof holds for any implementation of $\rho$ that satisfies those axioms, axioms which are notably verified by $\rho^{\mathsf{ms}}$.

Apart from such modeling tricks, defining and proving a fully-explicit game in SQUIRREL did not raise any particular challenges. We refer the interested reader to the full models [45] as well as the documentation of SQUIRREL for additional details. For a user familiar with SQUIRREL, we estimate that all proofs can be completed in roughly a few weeks of human-time work. The most time-consuming proof is the one for the billing case-study, mainly due to the size of the functionality. On a common laptop, SQUIRREL verifies most proofs in a couple of seconds, and takes 3 minutes for the billing case study, once again due to its number of case distinctions based on the many instructions in the functionality.

| | Functionality | Security | LoC Model | LoC Proof |
|---|---|---|---|---|
| **voting** | | $\mathcal{G}_{\mathsf{Vote-priv}}$ | | |
| | $\mathcal{F}^{\rho}_{\mathrm{VS}}$ [12] | ✘ | | |
| | $\mathcal{F}^{\rho,\mathsf{fix}}_{\mathrm{VS}}$ | ✔ | 146 | 400 |
| **lottery** | | $\mathcal{G}_{\mathsf{Lottery}}$ | | |
| | $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$ [13] | ✘ | | |
| | $\mathcal{F}^{\mathsf{Eligible},\mathsf{fix}}_{\mathsf{Anon-Selection}}$ | ✔ | 214 | 299 |
| | $\mathcal{F}^{\mathsf{LE}}_{\mathsf{Lottery}}$ [14] | ✔ | 200 | 157 |
| **SMPC** | | $\mathcal{G}_{\mathsf{SMPC}}$ | | |
| | $\mathcal{F}_{\mathsf{SMPC}}$[54], [55] | ✘[1] | | |
| | $[\mathcal{F}_{\mathsf{SMPC}}]^{\mathsf{ID}}_{\perp}$ [15], [16] | ✘ | | |
| | $[\mathcal{F}_{\mathsf{SMPC}}]^{\mathsf{ID},\mathsf{fix}}_{\perp}$ | ✔ | 156 | 121 |
| **billing** | | $\mathcal{G}_{\mathsf{Billing}}$ | | |
| | $\mathcal{F}_{\mathsf{BIL}}$ [3] | ✘ | | |
| | $\mathcal{F}^{\mathsf{fix}}_{\mathsf{BIL}}$ | ✔ | 521 | 2108 |

[1] *as trivially expected, the basic SMPC functionality does not provide identifiable aborts.*

Table I: Security of the ideal functionalities.

A green tick ✔corresponds to a proof in SQUIRREL, with models available at [45]. For each model, we provide an indicative number of line of codes, which includes comments and empty lines, split into the part of the file which models the protocol, and the part which contains the proof. A red tick ✘corresponds to a not so ideal functionality, where we either give an attack on the security property we formalized as a game, or show that the functionality can be realized by clearly insecure games.

## VII. LESSONS LEARNED

Through our inspection of five ideal functionalities' security, we have identified some common mistakes. We try to summarize these here, by discussing issues directly arising from our case-studies, as well as additional subtleties of the UC model. We emphasize that our work does not aim to discourage the use of ideal functionalities; they are an essential tool for expressing security properties that are outside the reach of game-based definitions and provide a very efficient tool for composable proofs and protocol design. However, we strive for a more formal approach to their design.

**Size of ideal functionalities.** As illustrated by the billing case study (Section V-C), the sheer size of some functionalities inherently precludes their effective verification by human inspection. When specifying such intricate functionalities, small errors seem difficult to avoid.

**Privacy properties are complex.** Privacy properties are notoriously subtle. For instance, it is well known that anonymity—where a party's name is never sent in clear over

the network—is not sufficient to guarantee privacy because the identity of the involved parties may leak implicitly through distinct behaviors (*e.g.*different error messages). As such, even in simple protocols, it is easy to make mistakes and introduce behaviors that compromise the privacy of agents. Ideal functionalities can actually be viewed as protocols meant to ensure a privacy property, which is then difficult to get right, as illustrated by the lottery case studies (Section V-A).

**Overly strong simulator.** Whenever the ideal functionality allows for direct interactions with the simulator, functionality designers must bear in mind the significant power this grants the simulator. Notably, as demonstrated in the e-voting case study (Section IV), the simulator might involuntarily be able to fix implementation issues within the protocol and conceal them. Essentially, any direct interaction between the simulator and the functionality cannot be observed by any of the honest agents. For instance, the billing functionality might be instantiated by protocols where the billing policy is fully private in the sense that consumers could not be informed of the terms under which they are being billed—a surprising situation—or by completely dishonest protocols where the billing policy invisibly varies between different consumers.

**No liveness.** The UC model does not offer any liveness guarantees: messages might not arrive, or never be read. So, in the basic model, an ideal functionality that allows the simulator to make direct queries to some procedure is useless. Indeed, any protocol that realizes the functionality without those queries also verifies the variant with them: the simulator for the former case is a valid simulator for the latter, simply by never calling the underlying procedures. Thus, one must rely on a round based execution mode for such cases, as done in the SMPC case-studies (Section V-B). For instance, we show that the $[_]_{\perp}^{\mathsf{ID}}$ transformation from [15] is useless without the corresponding round-based execution model. Such a change of execution model can be easily overlooked, leading to its misuse, and potentially yielding non-executable or insecure functionalities, as illustrated by the SMPC case-study.

**Single trust model.** When using an ideal functionality to capture a security property, one often models distinct properties simultaneously, such as privacy, eligibility and verifiability in the case of e-voting, as exemplified by the functionality $\mathcal{F}_{\mathrm{VS}}^{\rho}$ of Section IV-A. One should then keep in mind that the sub-properties are all verified under the same trust model defined by the ideal functionality. This can lead to an unsatisfying situation, as one often wants to verify those properties against distinct threat models.

## REFERENCES


[1] Swiss Federal Chancellery, "Federal Chancellery Ordinance on Electronic Voting (OEV)," 2022. [Online]. Available: https://www.fedlex.admin.ch/eli/cc/2022/336/en

[2] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *42nd Annual Symposium on Foundations of Computer Science, (FOCS 2001)*. IEEE Computer Society, 2001, pp. 136–145.

[3] A. Rial, G. Danezis, and M. Kohlweiss, "Privacy-preserving smart metering revisited," *International Journal of Information Security*, vol. 17, no. 1, pp. 1–31, Feb 2018.

[4] R. Canetti, P. Jain, M. Swanberg, and M. Varia, "Universally composable end-to-end secure messaging," in *Advances in Cryptology (CRYPTO 2022)*. Springer Nature Switzerland, 2022, pp. 3–33.

[5] A. Kiayias and O. S. T. Litos, "A composable security treatment of the lightning network," *IACR Cryptol. ePrint Arch.*, p. 778, 2019.

[6] M. Ciampi, A. Kiayias, and Y. Shen, "Universal composable transaction serialization with order fairness," Cryptology ePrint Archive, Paper 2024/1296, 2024.

[7] J. Camenisch, M. Drijvers, and A. Lehmann, "Universally composable direct anonymous attestation," in *Public-Key Cryptography (PKC 2016)*. Springer, 2016, pp. 234–264.

[8] J. Camenisch, M. Dubovitskaya, K. Haralambiev, and M. Kohlweiss, "Composable and modular anonymous credentials: Definitions and practical constructions," in *21st International Conference on Advances in Cryptology (ASIACRYPT 2015)*. Berlin, Heidelberg: Springer-Verlag, 2015, p. 262–288.

[9] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas, "Bitcoin as a transaction ledger: A composable treatment," *Journal of Cryptology*, vol. 37, no. 2, p. 18, 2024.

[10] R. Needham and M. Schroeder, "Using encryption for authentication in large networks of computers," *Communication of the ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[11] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1996)*, ser. LNCS, vol. 1055. Springer, 1996, pp. 147–166.

[12] A. Szepieniec and B. Preneel, "New techniques for electronic voting," *USENIX Journal of Election Technology and Systems (JETS)*, Aug. 2015.

[13] F. Baldimtsi, V. Madathil, A. Scafuro, and L. Zhou, "Anonymous lottery in the proof-of-stake setting," in *33rd IEEE Computer Security Foundations Symposium (CSF 2020)*, 2020, pp. 318–333.

[14] C. Ganesh, C. Orlandi, and D. Tschudi, "Proof-of-stake protocols for privacy-aware blockchains," in *38th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2019)*, ser. Lecture Notes in Computer Science. Springer, 2019, pp. 690–719.

[15] Y. Ishai, R. Ostrovsky, and V. Zikas, "Secure multi-party computation with identifiable abort," in *Advances in Cryptology (CRYPTO 2014)*, ser. Lecture Notes in Computer Science, vol. 8617. Springer, 2014, pp. 369–386.

[16] M. Simkin, L. Siniscalchi, and S. Yakoubov, "On sufficient oracles for secure computation with identifiable abort," in *13th International Conference on Security and Cryptography for Networks (SCN 2022)*, ser. Lecture Notes in Computer Science, vol. 13409. Springer, 2022, pp. 494–515.

[17] R. Cohen and Y. Lindell, "Fairness versus guaranteed output delivery in secure multiparty computation," *J. Cryptol.*, vol. 30, no. 4, pp. 1157–1186, 2017.

[18] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations among notions of security for public-key encryption schemes," in *Advances in Cryptology (CRYPTO 1998)*, ser. Lecture Notes in Computer Science, vol. 1462. Springer, 1998, pp. 26–45.

[19] M. Bellare, D. Hofheinz, and E. Kiltz, "Subtleties in the definition of IND-CCA: When and how should challenge decryption be disallowed?" *Journal of Cryptology*, vol. 28, no. 1, pp. 29–48, 2015.

[20] D. Bernhard, V. Cortier, D. Galindo, O. Pereira, and B. Warinschi, "Sok: A comprehensive analysis of game-based ballot privacy definitions," in *IEEE Symposium on Security and Privacy (S&P 2015)*, 2015, pp. 499–516.

[21] D. Baelde, S. Delaune, C. Jacomme, A. Koutsos, and S. Moreau, "An Interactive Prover for Protocol Verification in the Computational Model," in *42nd IEEE Symposium on Security and Privacy (S&P 2021)*, May 2021.

[22] R. Canetti, "Universally composable signature, certification, and authentication," in *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004)*, 2004, p. 219.

[23] ——, "A unified framework for analyzing security of protocols," *Electron. Colloquium Comput. Complex.*, vol. TR01-016, 2001.

[24] R. Canetti and H. Krawczyk, "Universally composable notions of key exchange and secure channels," in *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2002)*, ser. Lecture Notes in Computer Science, vol. 2332. Springer, 2002, pp. 337–351.

[25] R. Canetti and T. Rabin, "Universal composition with joint state," in *Advances in Cryptology (CRYPTO 2003)*, ser. Lecture Notes in Computer Science, vol. 2729. Springer, 2003, pp. 265–281.

[26] M. Backes and D. Hofheinz, "How to break and repair a universally composable signature functionality," in *7th International Conference on Information Security (ISC)*, ser. Lecture Notes in Computer Science, vol. 3225. Springer, 2004, pp. 61–72.

[27] S. Goldwasser, S. Micali, and R. L. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM J. Comput.*, vol. 17, no. 2, pp. 281–308, 1988.

[28] R. Canetti, H. Krawczyk, and J. B. Nielsen, "Relaxing chosen-ciphertext security," in *Advances in Cryptology (CRYPTO 2003)*, ser. Lecture Notes in Computer Science, vol. 2729. Springer, 2003, pp. 565–582.

[29] S. Delaune, S. Kremer, and O. Pereira, "Simulation based security in the applied pi calculus," in *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, (FSTTCS 2009)*, ser. LIPIcs, vol. 4, 2009, pp. 169–180.

[30] F. Böhl and D. Unruh, "Symbolic universal composability," in *26th Computer Security Foundations Symposium (CSF 2013)*. IEEE Computer Society, 2013, pp. 257–271.

[31] B. Blanchet, V. Cheval, and V. Cortier, "Proverif with lemmas, induction, fast subsumption, and much more," in *43rd IEEE Symposium on Security and Privacy (S&P 2022)*. IEEE, 2022, pp. 69–86.

[32] R. Canetti and J. Herzog, "Universally composable symbolic security analysis," *J. Cryptol.*, vol. 24, no. 1, pp. 83–147, 2011.

[33] R. Canetti, A. Stoughton, and M. Varia, "Easyuc: Using easycrypt to mechanize proofs of universally composable security," in *32nd IEEE Computer Security Foundations Symposium (CSF 2019)*. IEEE, 2019, pp. 167–183.

[34] R. Cohen, J. Doerner, E. Lee, A. Lysyanskaya, and L. Roy, "An unstoppable ideal functionality for signatures and a modular analysis of the dolev-strong broadcast," in *23rd International Conference of Theory of Cryptography (TCC 2025)*, ser. Lecture Notes in Computer Science. Springer, 2025.

[35] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," Cryptology ePrint Archive, Paper 2000/067, 2000, version 20200212:021048.

[36] C. Matt, J. B. Nielsen, and S. E. Thomsen, "Formalizing delayed adaptive corruptions and the security of flooding networks," in *Advances in Cryptology (CRYPTO 2022)*. Springer, 2022, pp. 400–430.

[37] C. Badertscher, R. Canetti, J. Hesse, B. Tackmann, and V. Zikas, "Universal composition with global subroutines: Capturing global setup within plain uc," in *Theory of Cryptography*. Springer, 2020, pp. 1–30.

[38] M. Backes, B. Pfitzmann, M. Waidner, and M. Steiner, "Polynomial liveness," *Journal of Computer Security*, 2004.

[39] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *25th International Conference on Computer Aided Verification (CAV 2013)*. Springer, 2013, pp. 696–701.

[40] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and proverif," *Foundations and Trends® in Privacy and Security*, vol. 1, no. 1-2, pp. 1–135, 2016.

[41] D. Baelde, S. Delaune, C. Jacomme, A. Koutsos, and S. Moreau, "An interactive prover for protocol verification in the computational model," in *42nd IEEE Symposium on Security and Privacy, (S&P 2021)*. IEEE, 2021, pp. 537–554.

[42] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Trans. Dependable Secur. Comput.*, vol. 5, no. 4, pp. 193–207, 2008.

[43] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *Advances in Cryptology (CRYPTO 2011)*, ser. Lecture Notes in Computer Science, vol. 6841. Springer, 2011, pp. 71–90.

[44] *The Rocq Prover Reference Manual*, 2025, release 9.0.0. [Online]. Available: https://rocq-prover.org/doc/V9.0.0/refman/index.html

[45] M. Arapinis, V. Cortier, H. de Groote, C. Jacomme, and S. Kremer, "Squirrel models," 2025. [Online]. Available: https://github.com/charlie-j/are-ideal-functionalities-really-ideal

[46] J. D. C. Benaloh, *Verifiable secret-ballot elections*. Yale University, 1987.

[47] J. C. Benaloh and M. Yung, "Distributing the power of a government to enhance the privacy of voters," in *5th annual ACM symposium on Principles of distributed computing (PODC 1986)*, 1986, pp. 52–62.

[48] R. Cleve, "Limits on the security of coin flips when half the processors are faulty (extended abstract)," in *18th Annual ACM Symposium on Theory of Computing (STOC 1986)*. ACM, 1986, pp. 364–369.

[49] S. Halevi, C. Hazay, A. Polychroniadou, and M. Venkitasubramaniam, "Round-optimal secure multi-party computation," Cryptology ePrint Archive, Paper 2017/1056, 2017.

[50] S. Goldwasser and Y. Lindell, "Secure computation without agreement," in *16th International Conference on Distributed Computing (DISC 2002)*, ser. Lecture Notes in Computer Science, vol. 2508. Springer, 2002, pp. 17–32.

[51] M. Simkin, L. Siniscalchi, and S. Yakoubov, "On sufficient oracles for secure computation with identifiable abort," Cryptology ePrint Archive, Paper 2021/151, 2021.

[52] M. Backes, J. Dreier, S. Kremer, and R. Künnemann, "A novel approach for reasoning about liveness in cryptographic protocols and its application to fair exchange," in *2nd IEEE European Symposium on Security and Privacy (EuroS&P'17)*. Paris, France: IEEE Computer Society, Apr. 2017, pp. 76–91.

[53] N. Brandt, S. Maier, T. Müller, and J. Müller-Quade, "Constructing secure multi-party computation with identifiable abort," *IACR Cryptol. ePrint Arch.*, p. 153, 2020.

[54] O. Goldreich, *Foundations of Cryptography: Basic Applications*. Cambridge University Press, 2004.

[55] S. Halevi, C. Hazay, A. Polychroniadou, and M. Venkitasubramaniam, "Round-optimal secure multi-party computation," in *Advances in Cryptology (CRYPTO 2018)*, 2018.


# APPENDIX A
# ELECTRONIC VOTING

## A. An explicit version of the $\mathcal{G}^{\mathcal{A},\Pi,\beta}_{\mathsf{Vote-priv}}$ security game

We have slightly adapted the $\mathcal{G}^{\mathcal{A},\Pi,\beta}_{\mathsf{Vote-priv}}$ security game to the fixed functionality, to account for the fact that the adversary is allowed to block some votes and this should not be considered as an attack. Hence we require that the attacker may see the result of the election only if the sets of *non blocked* votes are identical in both worlds. The resulting game $\mathcal{G}^{\mathcal{A},\Pi,\beta}_{\mathsf{Vote-priv-ext}}$ is presented in Figure 10.

Then, to be able to use SQUIRREL we have to slightly rewrite the security game, as the communication model differs from the UC framework. The resulting game is presented in Figure 11.

# APPENDIX B
# LOTTERY

## A. $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$ does not capture anonymity

The anonymous lottery functionality $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$ does not satisfy the anonymity definition specified in Section V-A1 and as such does not capture anonymity. Indeed, an attack consisting of a PPT Adversary $\mathcal{A}$ realises a non-negligible probability of distinction of the outputs of $\mathcal{G}^{0}_{\mathsf{Lottery},\mathcal{A},\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}}$ and $\mathcal{G}^{1}_{\mathsf{Lottery},\mathcal{A},\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}}$. Here is such an adversary $\mathcal{A}$:

- Send $\mathbf{P} = \{P_0, P_1, P_2\}$, and $\mathcal{H} = \{P_0, P_1\}$ to the Challenger.
- Send $\mathsf{HonestEligibilityCheck}(sid, \mathsf{t}, P_0, 0)$ to the Challenger where $\mathsf{t} \xleftarrow{\$} \mathbb{N}$ is any random tag.
- Send $\mathsf{HonestEligibilityCheck}(sid, \mathsf{t}, P_1, 1)$ to the Challenger.
- Send $\mathsf{HonestCreateProof}(sid, \mathsf{t}, m, P_0, P_1)$ to the Challenger where $m$ is any random message.

**Game** $\mathcal{G}^{\mathcal{A},\Pi,\beta}_{\mathsf{Vote-priv-ext}}$

- $\mathbf{V} \leftarrow \mathcal{A}$ with $\mathbf{V} = \{\mathsf{V}_1 \ldots, \mathsf{V}_n\}$ (the voters)
- $\mathcal{H}_{\mathbf{V}} \leftarrow \mathcal{A}$ with $\mathcal{H}_{\mathbf{V}} \subseteq \mathbf{V}$ (the honest voters)
- $\mathsf{A} \leftarrow \mathcal{A}$ (the authority)
- $\mathcal{C}$ initialises $\Pi(\mathbf{V}, \mathsf{A})$ with back-doors linked with $\mathcal{A}$
- $\mathcal{C}$ initialises empty lists $\mathsf{Voted}_0$, $\mathsf{Voted}_1$ and NoBlock which are part of the Oracles shared state
- $\mathcal{C}$ initializes the flag tallied to $\perp$ which is part of the Oracles shared state

Upon receiving $\mathsf{Guess}(sid, \beta^*)$ from $\mathcal{A}$
// $\mathcal{A}$'s guess received on $\mathcal{C}$'s backdoor tape
Output $\beta^*$

Upon receiving $\mathsf{HonestVote}(sid, V, v_0, v_1)$ from $\mathcal{A}$
If $V \in \mathcal{H}_{\mathbf{V}}$ and $\neg$tallied
Send $(\textsc{Vote}, sid, v_\beta)$ to $\Pi$ on behalf of $V$
$\mathsf{Voted}_0 \leftarrow \mathsf{Voted}_0@[(V, v_0)]$
$\mathsf{Voted}_1 \leftarrow \mathsf{Voted}_1@[(V, v_1)]$

Upon receiving $\mathsf{No-Block}(sid, V)$ from $\Pi$
NoBlock $\leftarrow$ NoBlock$@[V]$

Upon receiving $\mathsf{Tally}(sid)$ from $\mathcal{A}$
$\mathsf{NBVoted}_0 \leftarrow$ sublist of $\mathsf{Voted}_0$ such that voters are in NoBlock
$\mathsf{NBVoted}_1 \leftarrow$ sublist of $\mathsf{Voted}_1$ such that voters are in NoBlock
If $\rho^{\mathsf{ms}}(\mathsf{NBVoted}_0) = \rho^{\mathsf{ms}}(\mathsf{NBVoted}_1)$
Send $(\textsc{Tally}, sid)$ to $\Pi$ on behalf of A
tallied $= \top$

Figure 10: Updated game $\mathcal{G}^{\mathcal{A},\Pi,\beta}_{\mathsf{Vote-priv-ext}}$ for vote privacy. The modified parts are highlighted.

**Game** $\mathcal{G}^{\mathcal{A},\Pi,\beta}_{\mathsf{Vote-priv}}$ explicit

- $\mathbf{V} \leftarrow \mathcal{A}$ with $\mathbf{V} = \{\mathsf{V}_1 \ldots, \mathsf{V}_n\}$ (the voters)
- $\mathcal{H}_{\mathbf{V}} \leftarrow \mathcal{A}$ with $\mathcal{H}_{\mathbf{V}} \subseteq \mathbf{V}$ (the honest voters)
- $\mathsf{A} \leftarrow \mathcal{A}$ (the authority)
- $\mathcal{C}$ initialises $\Pi(\mathbf{V}, \mathsf{A})$ with back-doors linked with $\mathcal{A}$
- $\mathcal{C}$ initialises empty lists Voted, $\mathsf{Voted}_0$, $\mathsf{Voted}_1$ and NoBlock which are part of the Oracles shared state
- $\mathcal{C}$ initializes the flag tallied to $\perp$ which is part of the Oracles shared state

Upon receiving $\mathsf{Guess}(sid, \beta^*)$ from $\mathcal{A}$
// $\mathcal{A}$'s guess received on $\mathcal{C}$'s backdoor tape
Output $\beta^*$

Upon receiving $\mathsf{HonestVote}(sid, V, v_i, v_j)$ from $\mathcal{A}$
If $V$ is honest and tallied $= \perp$
Send $(\textsc{Vote}, sid, V)$ to $\mathcal{A}$
$\mathsf{Voted}_0 \leftarrow \mathsf{Voted}_0@[(V, v_0)]$
$\mathsf{Voted}_1 \leftarrow \mathsf{Voted}_1@[(V, v_1)]$

Upon receiving $\mathsf{CorruptedVote}(sid, V, v)$ from $\mathcal{A}$
If $V$ is not honest and tallied $= \perp$
Voted $\leftarrow$ Voted$@[(V, v)]$

Upon receiving $\mathsf{No-Block}(sid, V)$ from $\mathcal{A}$
NoBlock $\leftarrow$ NoBlock$@[V]$

Upon receiving $\mathsf{HonestTally}(sid)$ from $\mathcal{A}$
$\mathsf{NBVoted}_0 \leftarrow$ sublist of $\mathsf{Voted}_0$ such that voters are in NoBlock
$\mathsf{NBVoted}_1 \leftarrow$ sublist of $\mathsf{Voted}_1$ such that voters are in NoBlock
NBVoted $\leftarrow$ sublist of Voted such that voters are in NoBlock
If $\rho^{\mathsf{ms}}(\mathsf{NBVoted}_0) = \rho^{\mathsf{ms}}(\mathsf{NBVoted}_1)$
Calculate $t = \rho^{\mathsf{ms}}(\mathsf{NBVoted} \cup \mathsf{NBVoted}_\beta)$.
Send $(\textsc{Result}, sid, t)$ to $\mathcal{A}$
tallied $= \top$

Figure 11: An explicit version of the game based definition for vote-privacy. The modifications are highlighted.

- Upon receiving $(\textsc{Proof}, \mathsf{t}, m)$ from $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$ return $(\textsc{Done}, \psi, \mathsf{t}, m)$ to $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$ where $\psi \xleftarrow{\$} \Psi$ is any random proof.
- Send $(\textsc{Verify}, sid, \psi^*, \mathsf{t}, m)$ to $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$ on behalf of $P_2$ where $\psi^* \xleftarrow{\$} \Psi$ is any random proof $\psi^* \neq \psi$.
- Upon receiving $(\textsc{Verify}, sid, (\psi^*, \mathsf{t}, m))$ from $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$ return $w \leftarrow (P_0, \mathsf{t}, m)$ to $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$.
- Upon receiving $(\textsc{Verified}, sid, (\psi^*, \mathsf{t}, m), b)$ from $\mathcal{F}^{\mathsf{Eligible}}_{\mathsf{Anon-Selection}}$ on behalf of $P_2$ set $\beta^* \leftarrow 1 - b$.
- Send $\beta^*$ to the Challenger.

### B. Security of $\mathcal{F}^{\mathsf{LE}}_{\mathsf{Lottery}}$

We provide the definition of $\mathcal{F}^{\mathsf{LE}}_{\mathsf{Lottery}}$ in Figure 12, as defined in [14]. We provide the definition of $\mathcal{G}_{\mathsf{Lottery}}$ adapted to the interface of $\mathcal{F}^{\mathsf{LE}}_{\mathsf{Lottery}}$ in Figure 6.

**Functionality $\mathcal{F}_{\mathsf{Lottery}}^{\mathsf{LE}}$**

The functionality is parametrized by a set $\mathcal{E}$ of entries, and a predicate function $\mathsf{LE}$ which takes as input the relative stake (represented by a bit string in $\{0,1\}^{\ell_\alpha}$) and randomness (in $\{0,1\}^{\ell_{\mathsf{VRF}}}$).

**Registered Parties.** The functionality maintains a list $\mathcal{P}$ of registered parties. For each registered party the functionality maintains a message buffer.

**Stakeholders.** The functionality maintains a list of (the initial) stakeholders $\mathsf{pid}_1, \ldots, \mathsf{pid}_k$ and their relative stakes $\mathsf{pid}_1.\alpha, \ldots, \mathsf{pid}_k.\alpha$. Finally, the functionality also manages a table $T(,)$ where the entry $T(\mathsf{pid}, \mathsf{e})$ defines whether pid is allowed to publish a message relative to entry e. Initially, the table $T$ is empty.

- Upon receiving $(\textsc{Lottery}, sid, e)$ from a stakeholder pid (or the adversary in the name of corrupted participant pid) do the following:
  1) If $T(\mathsf{pid}, \mathsf{e})$ is undefined sample a random value $r \in \{0,1\}^\ell$ and set $T(\mathsf{pid}, \mathsf{e}) = \mathsf{LE}(\mathsf{pid}.\alpha; r)$.
  2) Output $(\textsc{Lottery}, sid, \mathsf{e}, T(\mathsf{pid}, \mathsf{e}))$ to pid (or the adversary).
- Upon receiving $(\textsc{Send}, sid, \mathsf{e}, m)$ from a stakeholder pid (or the adversary in the name of corrupted stakeholder pid) do the following:
  1) If $T(\mathsf{pid}, \mathsf{e}) = 1$ add $(\mathsf{e}, m)$ to the message buffers of all registered parties and output $(\textsc{Send}, sid, \mathsf{e}, m)$ to the adversary.
- Upon receiving $(\textsc{Send}, sid, \mathsf{e}, m, P')$ from the adversary in the name of corrupted stakeholder pid do the following:
  1) If $T(\mathsf{pid}, \mathsf{e}) = 1$ add $(\mathsf{e}, m)$ to the message buffer of $P'$ and output $(\textsc{Send}, sid, \mathsf{e}, m, P')$ to the adversary.
- Upon receiving $(\textsc{Fetch-New}, sid)$ from a party $P$ (or the adversary on behalf of $P$) do the following:
  1) Output the content of $P$'s message buffer and empty it.

**Get Information**

- Upon receiving $(\textsc{Get-Stake}, sid)$ from a stakeholder pid (or the adversary on behalf of pid) return $(\textsc{Get-Stake}, sid, \mathsf{pid}.\alpha)$.

Figure 12: Ideal functionality $\mathcal{F}_{\mathsf{Lottery}}^{\mathsf{LE}}$ as written in [14]

## APPENDIX C
## BILLING

### A. The ideal functionality of smart metering $\mathcal{F}_{\mathsf{BIL}}$

We reproduce in Figures 14 and 15 the ideal functionality $\mathcal{F}_{\mathsf{BIL}}$ proposed in [3] for defining the security of privacy-preserving smart metering protocols. The parts in red correspond to our fixed version $\mathcal{F}_{\mathsf{BIL}}^{\mathsf{fix}}$.

### B. Security game for billing

We propose a game-based definition $\mathcal{G}_{\mathsf{Billing}}$, defined in Figure 16, that captures the fact that:

- when all the meters are honest, a honest user should not be charged more than the price corresponding to their actual consumption,
- a dishonest user controlling some of their meters should not be able to pay less than the price charged by their honest meters.

The adversary wins if he can break one of these two properties.

### C. Attack on $\mathcal{F}_{\mathsf{BIL}}$

$\mathcal{F}_{\mathsf{BIL}}$ allows the adversary to choose the paid price as soon as they control one of the meters. It does not guarantee that the price set by the adversary includes at least the honest meters' readings. That is, a client with a dishonest meter might not pay anything even if some of their honest meters have reported some consumption. An environment and an adversary realising this attack are specified below, where we have set $U_y = \{Y = \{(c,t) \to 1\}\}$, $U_c = \mathbb{N}$, $U_t = \mathbb{N}$, and $U_{bp} = \{bp\}$ the universes of policies, consumptions, times, and billing periods, and $M_{max} = 2$ the maximum size for the meter lists.

$\mathcal{A}$:

- Send $\mathcal{V}$, $\mathcal{H}_\mathcal{V} \leftarrow \top$, $\mathcal{U} = \mathcal{U}_1$, $\mathcal{H}_\mathcal{U} \leftarrow \emptyset$, $\mathcal{M} = \{\mathcal{M}_1, \mathcal{M}_2\}$, and $\mathcal{H}_\mathcal{U} = \{\mathcal{M}_1\}$ to the Challenger.
- Send $(\mathsf{bil.policy.ini}, sid, bp, Y, \mathcal{V})$ to the Challenger.
- Upon receiving $(\mathsf{bil.policy.sim}, sid, bp, Y)$ from $\mathcal{F}_{\mathsf{BIL}}$ return $(\mathsf{bil.policy.rep}, sid, bp)$ to $\mathcal{F}_{\mathsf{BIL}}$.
- Send $(\mathsf{bil.listmeters.ini}, sid, bp, \mathcal{U}_1, \mathcal{M}_1, \mathcal{M}_2, \mathcal{V})$ to the Challenger.
- Upon receiving $(\mathsf{bil.listmeters.sim}, sid, ssid, \mathcal{U}_1)$ from $\mathcal{F}_{\mathsf{BIL}}$ return $(\mathsf{bil.listmeters.rep}, sid, ssid)$ to $\mathcal{F}_{\mathsf{BIL}}$.
- Upon receiving $(\mathsf{bil.listmeters.end}, sid, bp, \mathcal{M}_1, \mathcal{M}_2)$ from $\mathcal{F}_{\mathsf{BIL}}$ on behlaf of $\mathcal{U}_1$ do nothing (on $\mathcal{U}_1$'s backdoor).
- Send $(\mathsf{bil.consumption.ini}, sid, \mathcal{U}_1, bp, 1, 1, \mathcal{M}_1)$ to the Challenger.
- Upon receiving $(\mathsf{bil.consumption.sim}, sid, ssid, \mathcal{M}_1, \mathcal{U}_1)$ from $\mathcal{F}_{\mathsf{BIL}}$, return $(\mathsf{bil.consumption.rep}, sid, ssid)$ to $\mathcal{F}_{\mathsf{BIL}}$.
- Upon receiving $(\mathsf{bil.consumption.end}, sid, \mathcal{M}_1, bp, 1, 1)$ from $\mathcal{F}_{\mathsf{BIL}}$ on behlaf of $\mathcal{U}_1$ do nothing (on $\mathcal{U}_1$'s backdoor).

**Game** $\mathcal{G}^{\mathcal{A},\Pi,\beta}_{\mathsf{Lottery}}$

- $\mathcal{P} \leftarrow \mathcal{A}$ with $\mathcal{P} = \{P_1, \ldots, P_m\}$ (the registered parties)
- $\mathcal{H}_P \leftarrow \mathcal{A}$ with $\mathcal{H}_P \subseteq \mathcal{P}$ (the honest registered parties)
- $\mathcal{S} \leftarrow \mathcal{A}$ (the stakeholders) with $\mathcal{S} = \{(\mathsf{pid}_1, \mathsf{pid}_1.\alpha), \ldots, (\mathsf{pid}_n, \mathsf{pid}_n.\alpha)\}$ (the stakeholders)
- $\mathcal{H}_S \leftarrow \mathcal{A}$ with $\mathcal{H}_S \subseteq \{\mathsf{pid}_1, \ldots, \mathsf{pid}_n\}$ (the honest stakeholders)
- $\mathcal{C}$ initialises $\Pi$ with back-doors linked with $\mathcal{A}$
- $\mathcal{C}$ initialises empty tables $\mathsf{Eli}_0$ and $\mathsf{Eli}_1$ which are part of the Oracles shared state

Upon receiving $\mathsf{Guess}(sid, \beta^*)$ from $\mathcal{A}$
// $\mathcal{A}$'s guess received on $\mathcal{C}$'s backdoor tape
Output $\beta^*$

Upon receiving $\mathsf{HonestLottery}(sid, \mathsf{e}, \mathsf{pid}_i, \beta')$ from $\mathcal{A}$
If $\mathsf{pid}_i \in \mathcal{H}_S$
If $\beta' = 0$
Send $(\textsc{Lottery}, sid, \mathsf{e})$ to $\Pi$ on behalf of $\mathsf{pid}_i$
Wait for $(\textsc{Lottery}, sid, \mathsf{e}, b)$
If $\mathsf{Eli}_{\beta'}(\mathsf{pid}_i, \mathsf{e})$ is undefined
$\mathsf{Eli}_{\beta'}(\mathsf{pid}_i, \mathsf{e}) \leftarrow b$
Else // $\beta' = 1$
If $\mathsf{Eli}_{\beta'}(\mathsf{pid}_i, \mathsf{e})$ is undefined
Sample $r \in \{0,1\}^\ell$
$\mathsf{Eli}_{\beta'}(\mathsf{pid}_i, \mathsf{e}) \leftarrow \mathsf{LE}(\mathsf{pid}_i.\alpha; r)$

Upon receiving $\mathsf{HonestSend}(sid, \mathsf{e}, m, \mathsf{pid}_i, \mathsf{pid}_j)$ from $\mathcal{A}$
If $\mathsf{pid}_0, \mathsf{pid}_1 \in \mathcal{H}_S$ and $\mathsf{Eli}_0(\mathsf{pid}_0, \mathsf{e})$ and $\mathsf{Eli}_1(\mathsf{pid}_1, \mathsf{e})$ are defined and $\mathsf{Eli}_0(\mathsf{pid}_0, \mathsf{e}) = \mathsf{Eli}_1(\mathsf{pid}_1, \mathsf{e})$
Send $(\textsc{Send}, sid, \mathsf{e}, m)$ to $\Pi$ on behalf of $\mathsf{pid}_\beta$

Upon receiving $\mathsf{HonestFetchNew}(sid, P_i, \beta')$ from $\mathcal{A}$
If $P_i \in \mathcal{H}_P$ and $\beta' = \beta$
Send $(\textsc{Fetch-New}, sid)$ to $\Pi$ on behalf of $P_i$

Upon receiving $\mathsf{GetStake}(sid, \mathsf{pid}_i, \beta')$ from $\mathcal{A}$
If $P_i \in \mathcal{H}_P$ and $\beta' = \beta$
Send $(\textsc{Get-Stake}, sid)$ to $\Pi$ on behalf of $\mathsf{pid}_i$

Figure 13: Game $\mathcal{G}^{\mathcal{A},\Pi,\beta}_{\mathsf{Lottery}}$ for anonymous lottery $\mathcal{F}^{\mathsf{LE}}_{\mathsf{Lottery}}$.

- Send $(\mathsf{bil.period.ini}, sid, \mathcal{U}_1, bp, \mathcal{M}_1)$ to $\mathcal{F}_{\mathsf{BIL}}$.
- Upon receiving $(\mathsf{bil.period.sim}, sid, ssid, \mathcal{M}_1, \mathcal{U}_1)$ from $\mathcal{F}_{\mathsf{BIL}}$, return $(\mathsf{bil.period.rep}, sid, ssid)$ to $\mathcal{F}_{\mathsf{BIL}}$.
- Upon receiving $(\mathsf{bil.period.end}, sid, bp, \mathcal{M}_1, N)$ from $\mathcal{F}_{\mathsf{BIL}}$ on behlaf of $\mathcal{U}_1$ do nothing (on $\mathcal{U}_1$'s backdoor).
- Send $(\mathsf{bil.payment.ini}, sid, \mathcal{P}, bp)$ to $\mathcal{F}_{\mathsf{BIL}}$ on behalf of $\mathcal{U}_1$.
- Upon receiving $(\mathsf{bil.payment.sim}, sid, ssid, \mathcal{U}_1, \mathcal{P})$ from $\mathcal{F}_{\mathsf{BIL}}$ choose any $N_1$, $N_2$ (let's say $0, 0$) and send back $(\mathsf{bil.payment.rep}, sid, ssid, 0, \langle \mathcal{M}_1, N_1, \mathcal{M}_2, N_2 \rangle)$ to $\mathcal{F}_{\mathsf{BIL}}$.

$\mathcal{F}_{\mathsf{BIL}}$ 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. $\mathcal{F}_{\mathsf{BIL}}$ 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 $(sid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, ..., \mathcal{M}_{j_m}, 0)$.
   - 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 $\mathcal{M}_j$:
   - 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 $\mathcal{U}_i$.

Figure 14: Functionality $\mathcal{F}_{\mathsf{BIL}}$ : Interfaces bil.policy.$*$, bil.listmeters.$*$ and bil.consumption.$*$.

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, or 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 $(ssid, \mathcal{U}_i, \mathcal{P}, bp, Y')$.
   - 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 $(ssid, \mathcal{U}_i, \mathcal{P}, bp, Y)$ is not stored.
   - Delete $ssid$ from $(ssid, \mathcal{U}_i, \mathcal{P}, bp, Y)$.
   - Abort if $\mathcal{U}_i$ and $\mathcal{V}$ are corrupt and $m > M_{max}$.
   - If $\mathcal{U}_i$ or $\mathcal{V}$ are honest, retrieve $(\mathcal{M}_{j_1}, ..., M_{j_m})$ from the tuple $(sid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, ..., \mathcal{M}_{j_m}, 1)$, 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 $\mathcal{V}$ is honest ~~and the meters in the retrieved list are honest~~, set $p_{min} \leftarrow 0$ and for $k = 1$ to $m$, do the following:
     - If $\mathcal{M}_{j_k}$ is honest
       - 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 $M'_j = M_{j_k}$, $U'_i = U_i$ and $bp' = bp$.
       - For $n = 1$ to $N[\mathcal{M}_{j_k}, bp]$, set $p[\mathcal{M}_{j_k}, bp] = p[\mathcal{M}_{j_k}, bp] + Y(c[k, n], t[k, n])$.
       - $p_{min} \leftarrow p_{min} + p[\mathcal{M}_{j_k}, bp]$
   - If either $\mathcal{U}_i$ is honest or if $\mathcal{V}$ and the meters in the retrieved list are honest, set $p[bp] = p[\mathcal{M}_{j_1}, bp] + \cdots + p[M_{j_m}, bp]$, else if $p[bp] \geq p_{min}$ employ the value $p[bp]$ sent by $\mathcal{S}$, else abort.
   - 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 $\mathcal{P}$.

Figure 15: Functionality $\mathcal{F}_{\mathsf{BIL}}$: Interfaces bil.period.$*$ and bil.payment.$*$. Highlighted, our proposed fix, resulting in $\mathcal{F}_{\mathsf{BIL}}^{\mathsf{fix}}$. In particular, we removed the condition that required all meters to be honest when $\mathcal{U}_i$ is dishonest and we compute the price corresponding to the honest meters.

**Game** $\mathcal{G}^{\mathcal{A},\Pi}_{\mathsf{Billing}}$

- $\mathcal{V} \leftarrow \mathcal{B}$ (the provider)
- $\mathcal{H}_{\mathcal{V}} \leftarrow \mathcal{B}$ with $\mathcal{H}_{\mathcal{V}} \in \{\top, \bot\}$ (corruption status of provider)
- $\mathcal{U} \leftarrow \mathcal{B}$ with $\mathcal{U} = \{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$ (the users)
- $\mathcal{H}_{\mathcal{U}} \leftarrow \mathcal{B}$ with $\mathcal{H}_{\mathcal{U}} \subseteq \mathcal{U}$ (the honest users)
- $\mathcal{M} \leftarrow \mathcal{B}$ with $\mathcal{M} = \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ (the meters)
- $\mathcal{H}_{\mathcal{M}} \leftarrow \mathcal{B}$ with $\mathcal{H}_{\mathcal{M}} \subseteq \mathcal{M}$ (the honest meters)
- $\mathcal{C}$ initialises $\mathcal{A}$ and sends $\mathcal{V}, \mathcal{U}, \mathcal{M}, \mathcal{H}_{\mathcal{V}}, \mathcal{H}_{\mathcal{U}}$ and $\mathcal{H}_{\mathcal{M}}$ to $\mathcal{A}$
- $\mathcal{C}$ initialises $\Pi$ with back-doors linked with $\mathcal{A}$
- $\mathcal{C}$ initialises empty lists Policies, Meters, Consuptions which are part of the Oracles shared state
- $\mathcal{C}$ initialises flag $\beta \leftarrow 0$

Upon receiving (terminate.execution, $sid$) from $\mathcal{A}$
  // $\mathcal{A}$ terminates the execution of $\Pi$
  Output $\beta$

Upon receiving (bil.policy.ini, $sid, bp, Y, \mathcal{V}$) from $\mathcal{A}$
  If $\mathcal{H}_{\mathcal{V}} = \top$
    If $Y \in U_y$ and $bp \in U_{bp}$ and $(sid, bp, \star) \notin$ Policies
      Policies $\leftarrow$ Policies@$[(sid, bp, Y)]$
    Send (bil.policy.ini, $sid, bp, Y$) to $\Pi$ on behalf of $\mathcal{V}$

Upon receiving (bil.listmeters.ini, $sid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}, \mathcal{V}$) from $\mathcal{A}$
  If $\mathcal{H}_{\mathcal{V}} = \top$
    If $\mathcal{U}_i \in \mathcal{U}$ and $bp \in U_{bp}$ and $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m} \in \mathcal{M}$ and $(sid, bp, \mathcal{U}_i, \star, \ldots, \star) \notin$ Meters
      Meters $\leftarrow$ Meters@$[(sid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m})]$
    Send (bil.listmeters.ini, $sid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}$) to $\Pi$ on behalf of $\mathcal{V}$
  Else if $\mathcal{H}_{\mathcal{V}} = \bot$
    If $\mathcal{U}_i \in \mathcal{H}_{\mathcal{U}}$ and $bp \in U_{bp}$ and $\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m} \in \mathcal{M}$ and $(sid, bp, \mathcal{U}_i, \star, \ldots, \star) \notin$ Meters
      Meters $\leftarrow$ Meters@$[(sid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m})]$

Upon receiving (bil.consumption.ini, $sid, \mathcal{U}_i, bp, c, t, \mathcal{M}_j$) from $\mathcal{A}$
  If $\mathcal{M}_j \in \mathcal{H}_{\mathcal{M}}$
    If $bp \in U_{bp}$ and $c \in U_c$ and $t \in U_t$
      Consumptions $\leftarrow$ Consumptions@$[(\mathcal{M}_j, \mathcal{U}_i, bp, c, t)]$
    Send (bil.consumption.ini, $sid, \mathcal{U}_i, bp, c, t$) to $\Pi$ on behalf $\mathcal{M}_j$

Upon receiving (bil.period.ini, $sid, \mathcal{U}_i, bp, \mathcal{M}_j$) from $\mathcal{A}$
  If $\mathcal{M}_j \in \mathcal{H}_{\mathcal{M}}$
    If $\mathcal{U}_i \in \mathcal{U}$ and $bp \in U_{bp}$
      Send (bil.period.ini, $sid, \mathcal{U}_i, bp$) to $\Pi$ on behalf of $\mathcal{M}_j$

Upon receiving (bil.payment.ini, $sid, \mathcal{P}, bp, \mathcal{U}_i$) from $\mathcal{A}$
  If $\mathcal{U}_i \in \mathcal{H}_{\mathcal{U}}$
    Send (bil.payment.ini, $sid, \mathcal{P}, bp$) to $\Pi$ on behalf of $\mathcal{U}_i$

Upon receiving (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 $\mathcal{P}$ (from $\Pi$)
  If $\mathcal{P}$ is honest
    If $bp \notin U_{bp}$ or or $\mathcal{U}_i \notin \mathcal{U}$ or $m > M_{max}$ or ($\mathcal{H}_{\mathcal{V}} = \top$ and $(sid, bp, Y) \notin$ Policies)
      Set $\beta \leftarrow 1$
    Else
      If $\mathcal{V}$ is honest or $\mathcal{U}_i$ is honest
        If $(sid, bp, \mathcal{U}_i, \mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}) \in$ Meters
          $p_{min} \leftarrow 0$
          For all $\ell \in [\![m]\!]$ such that $\mathcal{M}_{j_\ell} \in \mathcal{H}_{\mathcal{M}}$
            If $N[m_{j_\ell}, bp] \neq |\{\!\{(\mathcal{M}_{j_\ell}, \mathcal{U}_i, bp, \star, \star) \in \text{Consumptions}\}\!\}|$
              Set $\beta \leftarrow 1$
            $p_{min} \leftarrow p_{min} + \sum_{(\mathcal{M}_{j_\ell}, \mathcal{U}_i, bp, c, t) \in \text{Consumptions}} Y(c, t)$
          If $p_{min} > p[bp]$
            Set $\beta \leftarrow 1$
          If $\{\mathcal{M}_{j_1}, \ldots, \mathcal{M}_{j_m}\} \subseteq \mathcal{H}_{\mathcal{M}}$ and $p_{min} \neq p[bp]$
            Set $\beta \leftarrow 1$
        Else
          Set $\beta \leftarrow 1$

Figure 16: Game $\mathcal{G}_{\mathsf{Billing}}$.