A Framework for the Sound Specification of Cryptographic Tasks

Juan A. Garay
AT&T Labs – Research
garay@research.att.com

Aggelos Kiayias
University of Connecticut
aggelos@cse.uconn.edu

Hong-Sheng Zhou
University of Connecticut
hszhou@cse.uconn.edu

March 12, 2010

Abstract

Nowadays it is widely accepted to formulate the security of a protocol carrying out a given task via the "trusted-party paradigm," where the protocol execution is compared with an ideal process where the outputs are computed by a trusted party that sees all the inputs. A protocol is said to securely carry out a given task if running the protocol with a realistic adversary amounts to "emulating" the ideal process with the appropriate trusted party. In the Universal Composability (UC) framework the program run by the trusted party is called an *ideal functionality*. While this simulation-based security formulation provides strong security guarantees, its usefulness is contingent on the properties and correct specification of the ideal functionality, which, as demonstrated in recent years by the coexistence of complex, multiple functionalities for the same task as well as by their "unstable" nature, does not seem to be an easy task.

In this paper we address this problem, by introducing a general methodology for the sound specification of ideal functionalities. First, we introduce the class of *canonical* ideal functionalities for a cryptographic task, which unifies the syntactic specification of a large class of cryptographic tasks under the same basic template functionality. Furthermore, this representation enables the isolation of the individual properties of a cryptographic task as separate members of the corresponding class. By endowing the class of canonical functionalities with an algebraic structure we are able to combine basic functionalities to a single final canonical functionality for a given task. Effectively, this puts forth a bottom-up approach for the specification of ideal functionalities: first one defines a set of basic constituent functionalities for the task at hand, and then combines them into a single ideal functionality taking advantage of the algebraic structure.

In our framework, the constituent functionalities of a task can be derived either directly or, following a translation strategy we introduce, from existing game-based definitions; such definitions have in many cases captured desired individual properties of cryptographic tasks, albeit in less adversarial settings. Our translation methodology entails a sequence of steps that systematically derive a corresponding canonical functionality given a game-based definition, effectively "lifting" the game-based definition to its composition-safe version.

We showcase our methodology by applying it to a variety of basic cryptographic tasks, including commitments, digital signatures, zero-knowledge proofs, and oblivious transfer. While in some cases our derived canonical functionalities are equivalent to existing formulations, thus attesting to the validity of our approach, in others they differ, enabling us to "debug" previous definitions and pinpoint their shortcomings.

Key words: Cryptographic protocols, security definitions, universal composability, lattices and partial orders.

Contents

1 Introduction

The Universal Composability (UC) framework proposed by Canetti [Can05], culminating a long sequence of simulation-based security definitions (cf. [GMW87, GL90, MR91, Bea91, Can00]; see also [PW01, BPW04, PS04, Küs06, CDPW07, LPV09] for alternative/extended frameworks), allows for arguing the security of cryptographic protocols in arbitrary settings where executions can be concurrent and adversarially interleaved. The framework is particularly attractive for the design of secure systems as it supports modularity, provides non-malleability across sessions [DDN00], and preserves security under composition.

In the UC framework, security is argued by providing a proof that a protocol realizes an *ideal functionality* $\mathcal{F}$ for the cryptographic task. While this simulation-based formulation provides compelling security guarantees, its usefulness is contingent on the properties of the realized ideal functionality. In particular, any ideal functionality is required to interact with an ideal-world adversary to whom it reveals aspects of its internal state. Thus, such a program can be quite far from an *idealization* of a given cryptographic task. To make things worse, the application of the framework to the analysis of many cryptographic schemes has shown that relatively complex ideal functionalities are the norm. This has frequently led to successive revisions of ideal functionality programs, the simultaneous coexistence of multiple different ideal functionalities for the same task, and the discovery of errors in their specification, which in turn would lead to flawed security guarantees for the protocols realizing them. A quick inspection of recent papers providing UC formulations of cryptographic tasks should suffice to support the claim about their complexity; see the initial treatment of digital signatures [Can01, BH04, Can04] for an example of need-to-revise and error-prone formulations of ideal functionalities.

In this paper we address this problem by introducing a general methodology for the sound specification of ideal functionalities. Following our methodology each task gives rise to a class of ideal functionalities that are consistent with the cryptographic task in terms of its actions. This representation unifies the syntactic specification of a large class of cryptographic tasks under the same basic template functionality, and, furthermore, it enables the isolation of the individual properties of a cryptographic task as separate members of the corresponding functionality class. This facilitates a fine-grain specification of the basic constituent properties of the ideal version of a task. At the same time, our methodology provides a way to combine constituent functionalities of a task to a single "supremum" ideal functionality that encompasses all constituent properties. This amounts to a *bottom-up approach* for achieving the original goal of expressing all required properties of a cryptographic task with a single functionality. This approach can be contrasted with the common "top-down" approach that specifies an ideal functionality capturing all essential properties at once, and then possibly relaxing it to bring it closer to realizability.

In addition, our methodology makes it easy to lift existing formulations of cryptographic properties for a task in case those have already been investigated in the form of *game-based definitions* to their corresponding UC counterparts. While such definitions are frequently easier to specify and understand as the appropriate formulations of the natural properties of the underlying cryptographic task, they typically provide a less satisfying level of security guarantees (as they may exclude composition, adaptive corruptions, non-malleability and other properties offered by the UC framework). Examples include the existential unforgeability notion for digital signatures [GMR88], IND-CPA security for public-key encryption, the hiding property of commitment schemes, and others.

Our results. We summarize our results in more detail:

1. *We introduce the notion of the class of* canonical functionalities *for a cryptographic task* T. Each member of this class has a simple, concise syntax built around two pass-through communication flows: one from the environment to the ideal-world adversary and another in the opposite direction. Every cryptographic task is associated to its corresponding canonical functionality class. Next, we define an operation over this class and show that the class has the algebraic structure of a semilattice which enables the joining of canonical functionalities. This algebraic structure is a unique feature that characterizes our bottom-up functionality specification approach. It imposes a natural ordering which enables the grading of canonical functionalities according to the level of security they offer, as well as the combination of more basic functionalities into a single final functionality for a task. Furthermore, the syntactic conciseness of our canonical functionalities gives rise to a well-defined communication (formal) language between the functionality and the other entities in an

ideal-world simulation which is instrumental in our methodology. These results are presented in Section 3.

We remark that the canonical functionality template abstracts all the common elements that are shared among ideal functionalities for many cryptographic tasks in the UC framework[1]. Once we have this formalism in place, this enables us to give very simple and concise formal definitions of the ideal functionalities for various different cryptographic tasks. When compared to the approach of defining functionalities of cryptographic tasks individually, our approach "contains" the complexity of such definitions in the template canonical functionality specification. This minimizes the effort of defining tasks in the UC framework without sacrificing rigor.

We further remark that while our canonical template and the algebraic structure helps understand the requirements for realizing a given functionality, it is not intended to be a tool for deciding the realizability of cryptographic tasks. Specifically, we prove that it is possible to draw a *realizability horizon* at some level of the semilattice of canonical functionalities that separates realizable functionalities from those that are impossible to realize (in the plain model). We note that setup assumptions (such as the existence of a CRS) can be viewed as modifiers to the level of the realizability horizon; specifically, as ways to increase the number of properties that can be achieved by a certain cryptographic task.

2. *We propose a new methodology for the sound writing of functionality programs.* We follow a bottom-up approach: we first define basic constituent functionalities for a cryptographic task and then we combine them taking advantage of the algebraic structure of the canonical functionality class. Next, we turn to the derivation of such constituent functionalities. These basic functionalities can be derived either *directly* or, following a translation strategy we introduce, from existing game-based definitions. Our game-to-functionality translation operates as follows. We divide games into two general types: *consistency games* and *hiding games*. The former capture properties such as correctness, unforgeability and binding, while the latter capture properties such as IND-CPA security and commitment hiding. Depending on the type of game, we present a sequence of steps that transform a game-based definition (whenever it exists or it is easily defined) into its corresponding canonical functionality. We demonstrate the soundness of this translation by showing that any scheme that realizes the resulting ideal functionality also possesses the properties offered by the game-based definition (Section 4).

3. *We showcase our methodology by applying it to a variety of basic cryptographic tasks.* We obtain ideal functionalities for digital signatures, oblivious transfer, commitments and zero-knowledge proofs of knowledge. In some cases (commitment, zero-knowledge) our derived canonical functionalities are equivalent to previously proposed functionalities in the literature, something that attests to the soundness of the bottom up approach; in others (signatures and oblivious transfer), our derived canonical functionalities differ from some previous definitions, allowing us to pinpoint their shortcomings. Specifically, in the case of signatures, we find that although the digital signature functionality presented in [Can04] in fact corresponds to the systematic transformation (using our methodology) of the game-based formulation of the Goldwasser *et al.* notion [GMR88], the attacker considered in [Can04] for the consistency game is more restricted than the one implied in [GMR88], which would result in a more relaxed functionality than the one actually presented there. In the case of oblivious transfer, the "debugging" goes beyond the specification of particular tasks, as in this case our methodology discovers a (fixable) structural inadequacy in the UC notion of "delayed output" in the ideal process. These results are presented in Section 5.

Related work. The work of Datta *et al.* [DDM+06] also relates individual properties of cryptographic tasks to sets of ideal functionalities. In their work each game-based property of a cryptographic task is mapped to a syntactic expression in process calculus that the ideal functionality program needs to satisfy. Based on this, they were able to generalize impossibility results on the realizability of specific cryptographic tasks such as bit commitment and group signatures. In contrast, while the current work also abstracts the basic properties of a cryptographic task in a syntactic fashion, it further provides a methodology to explicitly (and easily) write ideal functionality programs

[1] In this version we concentrate on deterministic ideal functionalities, which can formalize many cryptographic tasks such as digital signatures, public-key encryption, oblivious transfer, zero-knowledge and others.

within the UC framework that provably satisfy such properties.

Although in a different context, the work of Backes *et al.* [BPW04] also considers the transformation of an arbitrary consistency (called integrity in [BPW04]) property into a (low-level) ideal functionality that enjoys that property. As a general methodology, distinguishing features of our approach include applicability to arbitrary security properties (i.e., also hiding, not just consistency properties), being able to obtain algorithm-independent functionalities, and the ability to treat individual security properties separately, together with a mechanism to combine the derived functionalities into one end result.

Differences with the previous version of this report [GKZ09]. As mentioned above, our specification approach is based on a well-defined communication language between ideal functionalities and the other ideal-world entities, with tasks' consistency properties essentially being mapped to classes of strings in that language. In turn, these strings (in fact, "bad" strings, which correspond to properties not being satisfied) are to be recognizable within complex executions where symbols may be interleaved with other symbols. Our previous approach was not complete in the sense of capturing all possible interleaving scenarios, something that we address here (See Section 4.2.1).

The semilattice structure of our canonical functionalities allows us to draw a "realizability horizon," separating functionalities that can be realized (in the plain model) from those that cannot. We illustrate such a semilattice with the case of the commitment task, known to be unrealizable. The current version sheds additional light on this issue, by containing protocols realizing the functionalities resulting from the pair-wise combination of the task's properties (hiding, binding, and correctness). (See Figure 3 and Section 5.3.)

Comparisons of our derived functionalities, in particular for signatures and oblivious transfer, with corresponding previous functionalities has been expanded, as well as comparisons with additional related work, and proofs have been added. Finally, the change of title emphasizes our intent of presenting and promoting this work as a methodology for the (sound) specification of cryptographic tasks.

2 Preliminaries

We will lay out our results following the Universal Composability framework of Canetti [Can05]. Recall that in the UC framework, the environment is creating processes which are entities maintaining state across actions. Further, a cryptographic task T is associated with an ideal functionality which is a stateful entity. The functionality $\mathcal{F}_T$ is a "packaging" of the actions of the task T together with data fields that are persistent across action invocations. For example, an ideal functionality for the commitment task offers two actions, commit and open, and has a persistent data field that is generated by the commit action and used by the corresponding open action (the decommitment information). Similarly, an ideal functionality for the digital signature task offers three actions, key-generation, signature-generation, and signature-verification, and has a persistent data field that is updated by the signing action and used by the verification action (this is the list of messages that have been signed).

Given that the environment is generating all actions for the ideal functionality of a task, not all sequences of actions are sensible, as the environment is assumed adversarial. Actions that are deemed inconsistent with the current state are ignored by the ideal functionality. This implicitly determines a notion of "well-formedness" of action sequences that we will formalize in the next section. Furthermore, the ideal functionality may generate output to a party depending on its internal state and the influence of the adversary; to formalize the interaction between the functionality and the adversary with respect to output generation we will define in the next section two functions that we call the "public output" and "secret output" of the functionality.

Our definitions of well-formedness and public/secret output for a functionality will also require basic string operations. Given a sequence w consisting of elements from an alphabet $\Sigma = \{\mathtt{a}_1, \ldots, \mathtt{a}_k\}$, we let w_i denote the i-th element in w. We can obtain a *subsequence* of w, call it w', by erasing some of the elements in w without disturbing the relative positions of the remaining elements. We denote this by $w' \preccurlyeq w$ and we remark that $\epsilon \preccurlyeq w$ for any string w (here ϵ is the empty string). For a given set of strings S we define $S^{\preccurlyeq} = \{w' \mid \exists w \in S : w' \preccurlyeq w\}$. For any $w \in \Sigma^*$ we write $w' \prec w$ when w' is derived from w after substituting at least one symbol of w with the special symbol "$-$".

A *monoid* $(A, +)$ is a semigroup with an identity element. Any monoid possesses a preorder relation denoted by $\lesssim$ such that $a \lesssim b$ iff $\exists c : a + c = b$.

3 Canonical Ideal Functionalities

In this section we provide an explicit syntax for a class of functionalities that idealize the cryptographic task T — this is the *class of canonical functionalities* for the cryptographic task T. In this first formulation of canonical functionalities we focus on a wide class of cryptographic tasks whose action outputs are not required to follow an ideal probability distribution, i.e., the ideal functionality can be deterministic. Such tasks include digital signatures, commitment, public-key encryption, secure message transmission, zero-knowledge proofs, secure deterministic function evaluation, etc.

3.1 The communication language of ideal functionalities

We start by specifying the language of the communication between the ideal functionality $\mathcal{F}_T$ of a task T and the environment. The alphabet over which the environment communicates with the ideal functionality is parameterized by the security parameter $\lambda \in \mathbb{N}$ and is a finite set of symbols of the form $(\text{ACTION}, \mathbf{P}, x)$; note that we will usually omit reference to λ for simplicity. Here ACTION is a label that determines the action the environment instructs the functionality to do (e.g., COMMIT, SIGN, etc.). $\mathbf{P}$ is a tuple that designates the identifiers of the entities and their roles in the particular action, in particular which parties provided the input to the action and which parties should receive output. To differentiate multiple invocations of the functionality by the same group of entities, $\mathbf{P}$ may also include a session identifier sid. Finally, the value x is an encoding of the input to the action whose length is polynomial in λ; note that whenever $x = \epsilon$, we will drop x from the symbol notation for ease of reading.

In response to a symbol $(\text{ACTION}, \mathbf{P}, x)$, the ideal functionality may return a symbol $(\text{ACTIONRETURN}, \mathbf{P}, y)$ to some party (see below in what circumstances). The set of all symbols of the form $(\text{ACTION}, \mathbf{P}, x)$ and $(\text{ACTIONRETURN}, \mathbf{P}, y)$, constitutes the finite I/O alphabet of the ideal functionality, i.e., the communication alphabet between the functionality and the environment, and is denoted by Σ_T.

As mentioned in Section 2, the actions of a cryptographic scheme make sense only in certain order; for this reason not all strings over Σ_T are valid as action sequences. To formalize this, we associate with the ideal functionality the predicate WF_T called the *well-formedness predicate*. For any string $w \in (\Sigma_T)^*$ and symbol $\mathtt{a} \in \Sigma_T$, the well-formedness predicate $\mathsf{WF}_T(w, \mathtt{a})$ decides whether the string $w\mathtt{a}$ is sensible with respect to $\mathcal{F}_T$.

We also mentioned before that the ideal functionality may produce output based on its internal state and the current action symbol following the influence of the adversary. In order to enable an informed influence the adversary needs to know some information about the internal state of the functionality. This will be formally captured by a polynomial-time string mapping PO_T called the *public output* mapping, that given a string $w \in (\Sigma_T)^*$ and a symbol $\mathtt{a} \in \Sigma_T$ that satisfies $\mathsf{WF}_T(w, \mathtt{a})$, may return a value that is a part of the suggested output of the ideal functionality on action symbol $\mathtt{a}$ given the history w. For example, in the case of a zero-knowledge task $T = \mathtt{ZK}$, upon receiving $(\text{PROVE}, \langle P, V, sid\rangle, \langle x, m\rangle)$, $\mathsf{PO}_{\mathtt{ZK}}$ will output the pair $\langle x, \phi\rangle$ where $\phi = 1$ if and only if $\langle x, m\rangle$ belongs to the relation that parameterizes the zero-knowledge functionality. Depending on the task, the adversary may not need to know the correct output to influence the functionality. To capture this, another string mapping SO_T called the *secret output* mapping is defined. For example, for the task $T = \mathtt{OT}$ of oblivious transfer, upon receiving symbols $(\text{TRANSFER}, \langle S, R, sid\rangle, \langle m_0, m_1\rangle)$ and $(\text{TRANSFER}, \langle S, R, sid\rangle, i)$, the mapping $\mathsf{SO}_{\mathtt{OT}}$ will return m_i and the adversary will not need to know this value to influence the functionality to return it. On the other hand, in some cases, the ideal functionality will not suggest any output, i.e., the $\mathsf{SO}_T, \mathsf{PO}_T$ mappings will be empty, and then the adversary will be entirely responsible for the choice of output to the parties. For example, in the digital signature task $T = \mathtt{SIG}$, on action symbol $(\text{KEYGEN}, \langle S, sid\rangle)$ the mappings $\mathsf{PO}_{\mathtt{SIG}}$ and $\mathsf{SO}_{\mathtt{SIG}}$ will output the empty string as there is no need for the functionality to suggest any distribution on the public key pk that is returned in the symbol $(\text{KEYGENRETURN}, \langle S, sid\rangle, pk)$ and is selected by the adversary.

This completes the description of the I/O language used in the communication between the environment and the ideal functionality. $\mathcal{F}_T$ also communicates with an ideal world entity, called the ideal world adversary $\mathcal{S}$.

This interaction defines another communication language that is not bound by the alphabet of the real world. We next define this language formally. For each input action symbol $(\text{ACTION}, \mathbf{P}, x)$, the ideal functionality may want to notify the ideal world adversary. We capture this by introducing a set of "leaking-action" symbols $(\text{LEAKACTION}, \mathbf{P}, x')$ where x' will have a functional dependency on x according to the program of the ideal functionality. The public output of the ideal functionality (as defined by the PO_T mapping) will also be sent out with the LEAKACTION symbol to the adversary.

Conversely, the ideal world adversary may also communicate with the ideal functionality; to capture this interaction we introduce the "influence-action" symbols denoted by $(\text{INFLACTION}, \mathbf{P}, \cdot)$. Such symbols are used by $\mathcal{S}$ to influence the output of a particular action that is currently under-way. Moreover, the adversary may inform the functionality that a certain party is corrupted; for this purpose symbols of the form $(\text{CORRUPT}, \cdot)$ will be used. Finally, the adversary can control the way the ideal functionality handles corrupted parties through the use of $(\text{PATCH}, \cdot)$ symbols which enable the adversary to specify the inputs of corrupted parties as well as their state with respect to all currently ongoing actions. The exact syntax of $(\text{CORRUPT}, \cdot)$, $(\text{PATCH}, \cdot)$ symbols is explained below. The extended communication alphabet of the ideal functionality is denoted by Σ_T^{ext} and includes all I/O symbols of Σ_T as well as the corresponding INFLACTION, LEAKACTION, CORRUPT and PATCH symbols.

So far we introduced a syntax for the communication language between the ideal functionality and the other entities of the ideal world. Next, we will describe a well-defined class of structured ideal functionalities for a cryptographic task. We call this the class of *canonical functionalities* for the task.

3.2 The canonical functionality of a cryptographic task

The functionalities that we consider in this paper adhere to the following design choices: (i) the adversary is notified of all input actions (by means of LEAKACTION symbols), (ii) the ideal functionality produces output only after being instructed by the adversary through an INFLACTION symbol (this captures the "delayed output" property of [Can05]), (iii) the ideal functionality is a deterministic TM, and (iv) outputs are always given sequentially[2].

A canonical functionality is essentially defined by two functions: $\mathsf{suppress}()$ and $\mathsf{validate}()$. As stated in (i) above, given an action symbol $(\text{ACTION}, \mathbf{P}, x)$ a canonical ideal functionality will always notify the adversary about this input. The function $\mathsf{suppress}()$ will determine what information about x the ideal world adversary will learn. The output of $\mathsf{suppress}()$ will be passed into the LEAKACTION symbol. Specifically, the $\mathsf{suppress}()$ function is defined over $(\text{ACTION}, \mathbf{P}, x)$ symbols and will output some $x' \prec x$. We require that the $\mathsf{suppress}()$ function will always substitute with "$-$" the same locations of x, independently of x.

Together with a LEAKACTION symbol the adversary will also be given the public output of the ideal functionality. Furthermore, the canonical functionality will return output to some party whenever it receives an INFLACTION symbol. In particular given $(\text{INFLACTION}, \mathbf{P}, y')$ it will return $(\text{ACTIONRETURN}, \mathbf{P}, y)$ to the party in $\mathbf{P}$ that is supposed to receive output, where y consists of y' (which may be empty) concatenated with the secret output of the functionality (which may also be empty).

Given that not all output influences are consistent with the intended security properties of the cryptographic task, the $\mathsf{validate}()$ predicate is defined over strings of Σ^* and determines when the canonical functionality will halt. We note that $\mathsf{suppress}()$ is history independent while $\mathsf{validate}()$ is not. The intuition is that the $\mathsf{suppress}()$ function abstracts what the adversary learns about the possibly private inputs of parties (i.e., it captures the hiding aspects of the functionality) whereas the $\mathsf{validate}()$ predicate makes sure that the outputs produced by the functionality are consistent with its history according to the intended consistency properties of the task.

In order to perform the $\mathsf{validate}()$ check a canonical functionality needs to be stateful. The state of the functionality, denoted by $\mathsf{history}$, is the sequence of all I/O symbols ordered chronologically as received from and sent to the environment. We use $\mathsf{history}_{P_j}$ to denote all symbols associated to party P_j in $\mathsf{history}$, i.e., the ACTION symbols that were provided by P_j and ACTIONRETURN symbols that were returned to P_j.

[2] These design choices exclude certain aspects of functionalities from the present formulation, such as fairness, guaranteed output delivery and "object-oriented insulation" (i.e., the aspect of a functionality that prevents the adversary from knowing that an honest party makes a certain action, and the ability of the functionality to sample random outputs from given distributions; see, e.g., the treatment of digital signatures in [Can05, Pat05]). We stress that it is possible to integrate these aspects into our framework, which we defer to future work.

The CORRUPT and PATCH symbols are used to handle the behavior of corrupted parties. When a party P_j is corrupted, we allow the adversary to learn $\mathsf{history}_{P_j}$[3]. Moreover, to handle adaptive corruptions, we allow the adversary to rewrite the history of corrupted parties using the PATCH symbols in the following manner: a certain symbol that was provided by a corrupted party can be modified provided this symbol has not contributed to the view of any honest party. To facilitate this checking the canonical functionality will use an array called $\mathsf{binding}[\cdot]$ that for each symbol in history, records the set of honest parties whose view could have been affected by that symbol. In particular each time an honest party P_j receives output from the ideal functionality all previous symbols in history are marked with P_j in the $\mathsf{binding}[\cdot]$ table (binding all symbols is suitable as an instance of our canonical functionality that is supposed to capture a single session only).

We now have all the elements to present the exact formulation of the class of canonical functionalities $\mathscr{F}_T$. Each member of the class is specified by a pair of functions $\mathsf{suppress}(), \mathsf{validate}()$ as defined above. Specifically, the functionality responds to $msg = (\text{ACTION}, \mathbf{P}, x)$ symbols by issuing $(\text{LEAKACTION}, \mathbf{P}, x')$ symbols to the adversary where $x' = \mathsf{suppress}(msg)$ and to $msg = (\text{INFLACTION}, \mathbf{P}, y)$ from the adversary by issuing $(\text{ACTIONRETURN}, \mathbf{P}, y)$ symbols to the parties specified in $\mathbf{P}$ as long as $\mathsf{validate}(\mathsf{history}||msg) = 1$. Additionally it responds to CORRUPT, PATCH symbols as one would expect: given a $\text{CORRUPT}(P_j)$ symbol, it returns to the adversary the history of party P_j and given a $\text{PATCH}(\mathsf{history}')$ it modifies its internal state with respect to corrupted parties following $\mathsf{history}'$. A pictorial representation can be found in Figure 1, and a detailed description is given in Figure 2.

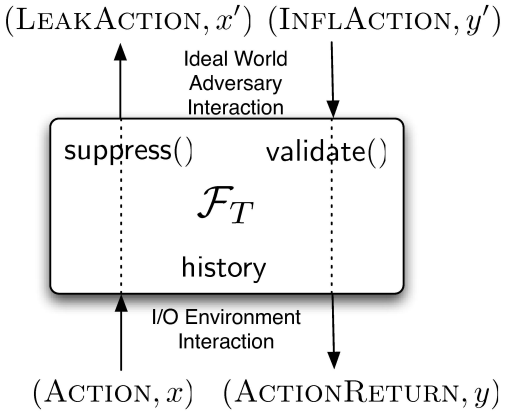

Figure 1: The canonical functionality: communication flows with the environment and adversary.

A canonical functionality defines a language over the symbols that are used by the functionality to communicate with the environment. We formalize this language as follows:

Definition 3.1. *Given a canonical functionality $\mathcal{F}_T$, an environment $\mathcal{Z}$ and an adversary $\mathcal{S}$, we define* $L_{\mathcal{F}_T,\mathcal{Z},\mathcal{S}} = \{w | w \in (\Sigma_T)^*$ such that w is equal to history of $\mathcal{F}_T$ in an execution with $\mathcal{Z}$ and $\mathcal{S}\}$.

Essentially the above language contains all possible execution histories. Since $\mathcal{Z}, \mathcal{S}$ are PPT there may be a different execution history for each choice of the random tapes of these machines. We may quantify the language over all possible environments $\mathcal{Z}$ and ideal world adversaries $\mathcal{S}$ in which case we will omit referencing them. Moreover, we may consider only those strings in history of $\mathcal{F}_T$ for which the environment $\mathcal{Z}$ returns 1; we will denote this ("bad") language as $B_{\mathcal{F}_T,\mathcal{Z},\mathcal{S}}$.

3.3 The algebraic structure of canonical functionality classes

The $\mathsf{suppress}(), \mathsf{validate}()$ parameterization effectively gives a range of canonical functionalities with security and correctness properties of different strength for the same cryptographic task. We next endow this class with an

[3]We also defer the treatment of forward security for now.

Canonical Functionality $\mathcal{F}_T^{\mathsf{suppress},\mathsf{validate}}$

Initially, $\mathsf{history} := \epsilon$ and $\mathsf{binding} := \epsilon$.

- Upon receiving $msg = (\text{ACTION}_i, \mathbf{P}, x)$ from some party P_j, if P_j is corrupted set $x' \leftarrow x$ else compute $x' \leftarrow \mathsf{suppress}(msg)$, set $msg' \leftarrow (\text{LEAKACTION}_i, \mathbf{P}, x')$. If $\mathsf{WF}_T(\mathsf{history}_{P_j}, msg) = 1$ then do the following:
  Send $\langle msg', \mathsf{PO}_T(\mathsf{history}, msg)\rangle$ to the adversary $\mathcal{S}$;
  record msg in history and also if P_j is uncorrupted, set $\mathsf{binding}[|\mathsf{history}|] = \{P_j\}$.
  Otherwise ($\mathsf{WF}_T(\mathsf{history}_{P_j}, msg) = 0$) ignore msg.
- Upon receiving $msg = (\text{INFLACTION}_i, \mathbf{P}, y')$ from the adversary $\mathcal{S}$, infer P_j from $\mathbf{P}$, and set $msg' \leftarrow (\text{ACTIONRETURN}_i, \mathbf{P}, y)$, where $y \leftarrow y'||\mathsf{SO}_T(\mathsf{history}, msg)$. If $\mathsf{WF}_T(\mathsf{history}_{P_j}, msg') = 1$ do the following:
  If $\mathsf{validate}(\mathsf{history}||msg') = 1$, then append msg' to history and send msg' to party P_j;
  if P_j is honest, set $\mathsf{binding}[k] \leftarrow \mathsf{binding}[k] \cup \{P_j\}$, $1 \le k \le |\mathsf{history}|$.
  Otherwise ($\mathsf{validate}(\mathsf{history}||msg') = 0$), if P_j is honest send an error symbol to P_j and halt.
  Otherwise ($\mathsf{WF}_T(\mathsf{history}_{P_j}, msg') = 0$ or P_j is corrupted) ignore msg.
- Upon receiving $msg = (\text{CORRUPT}, P_j)$ from the adversary $\mathcal{S}$, mark P_j as corrupted, return $\mathsf{history}_{P_j}$ to $\mathcal{S}$, and set $\mathsf{binding}[k] \leftarrow \mathsf{binding}[k] \setminus \{P_j\}$, $1 \le k \le |\mathsf{history}|$.
- Upon receiving $msg = (\text{PATCH}, \mathsf{history}')$ from the adversary $\mathcal{S}$ where $\mathsf{history}' \in (\Sigma_T)^{|\mathsf{history}|}$ do the following: if $\mathsf{binding}[k] = \emptyset$ set $\mathsf{history}[k] \leftarrow \mathsf{history}'[k]$, $1 \le k \le |\mathsf{history}|$.

Figure 2: Definition of the class of canonical functionalities $\mathscr{F}_T$ for a task T quantifying over all admissible pairs $\mathsf{suppress}(), \mathsf{validate}()$.

algebraic structure that will be helpful in classifying and combining the various canonical functionalities for the same cryptographic task.

We define a conjunction operation denoted by $\wedge$ on the class of canonical functionalities for a task T. This operation will enable us to combine canonical functionalities for a task T, while providing a concise way of representing members of the class in terms of "simpler" members. Observe that for any two members of the canonical functionality class that are parameterized by the functions $\mathsf{suppress}_1$ and $\mathsf{suppress}_2$, respectively, for any symbol $\mathtt{a} = (\text{ACTION}_i, \mathbf{P}, x)$, it holds that $\mathsf{suppress}_1((\text{ACTION}_i, \mathbf{P}, \mathsf{suppress}_2(\mathtt{a}))) = \mathsf{suppress}_2((\text{ACTION}_i, \mathbf{P}, \mathsf{suppress}_1(\mathtt{a})))$. This fact will be handy in the definition below.

Definition 3.2 (Conjuncting Functionalities)**.** *Given $\mathcal{F}_1 = \mathcal{F}_T^{\mathsf{suppress}_1,\mathsf{validate}_1}, \mathcal{F}_2 = \mathcal{F}_T^{\mathsf{suppress}_2,\mathsf{validate}_2} \in \mathscr{F}_T$ we define the* conjunction *$\mathcal{F}_1 \wedge \mathcal{F}_2$ of the two functionalities as the functionality $\mathcal{F}_T^{\mathsf{suppress},\mathsf{validate}} \in \mathscr{F}_T$, where (1) for any $\mathtt{a} = (\text{ACTION}, \mathbf{P}, x) \in \Sigma$, $\mathsf{suppress}(\mathtt{a}) = \mathsf{suppress}_1(\text{ACTION}_i, \mathbf{P}, \mathsf{suppress}_2(\mathtt{a}))$, and (2) $\mathsf{validate}() = \mathsf{validate}_1() \wedge \mathsf{validate}_2()$, i.e., the logical conjunction of the two validate predicates of $\mathcal{F}_1, \mathcal{F}_2$.*

We next show that the canonical functionality class for a task T has a monoid structure with identity element a canonical functionality that we call the *dummy functionality* for T defined as follows:

Definition 3.3 (Dummy Functionality)**.** *We call the canonical functionality $\mathcal{F}_T^{\text{dum}} \in \mathscr{F}_T$* dummy *if (1) for all x and any* ACTION*, $\mathsf{suppress}((\text{ACTION}, \mathbf{P}, x)) = x$, and (2) $\mathsf{validate}() = 1$ always.*

Observe that the dummy functionality does not capture any of the intended correctness or security properties of the cryptographic task T. This means that any protocol π UC-realizing $\mathcal{F}_T^{\text{dum}}$ will merely syntactically match the purpose of T but will lack any useful property.

Proposition 3.4. *$(\mathscr{F}_T, \wedge)$ is a commutative monoid with the dummy functionality $\mathcal{F}_T^{\text{dum}}$ as the identity element.*

Any commutative monoid has an associated preordering relation denoted by $\lesssim$; in the case of $(\mathscr{F}_T, \wedge)$ we say that $\mathcal{F}_1 \lesssim \mathcal{F}_2$ if and only if there exists $\mathcal{F}_3$ such that $\mathcal{F}_2 = \mathcal{F}_1 \wedge \mathcal{F}_3$. The intuitive interpretation of $\mathcal{F}_1 \lesssim \mathcal{F}_2$ is that $\mathcal{F}_2$ is at least as strict as $\mathcal{F}_1$ from a security point of view.

$\mathscr{F}_T$ together with $\wedge$ forms a bounded (join-)semilattice, i.e., every set of elements in $\mathscr{F}_T$ has a least upper bound. Note that (1) we use $\wedge$ in place of the standard $\vee$ in lattice theory as it is more consistent as an operator in our setting where lattice elements would capture security properties (and going higher in the lattice means that security increases), and (2) given that $(\mathscr{F}_T, \wedge)$ as a commutative monoid lacks the antisymmetric property, the semilattice would be in fact over the quotient $\mathscr{F}_T/\asymp$ where $\asymp$ is the equivalence relation defined as $\mathcal{F}_1 \asymp \mathcal{F}_2$ iff $\mathcal{F}_1 \lesssim \mathcal{F}_2$ and $\mathcal{F}_2 \lesssim \mathcal{F}_1$.

We next define the *top canonical functionality* $\mathcal{F}_T^{\text{top}}$ which is the supremum of all canonical functionalities in $\mathscr{F}_T$ representing the most stringent idealization of a cryptographic task. The top functionality suppresses all inputs (but allows the adversary to see the length of them), while it restricts all output influences of the adversary.

Definition 3.5 (Top Functionality)**.** *We call the canonical functionality $\mathcal{F}_T^{\text{top}} \in \mathscr{F}_T$* top *if for any* ACTION*, (1) for all x,* $\mathsf{suppress}((\text{ACTION}, \mathbf{P}, x)) = (-)^{|x|}$*, and (2) for all w, a,* $\mathsf{validate}(w\mathsf{a}) = 0$*.*

While the top functionality offers security it does so at the expense of providing no response to the parties that employ it, i.e., it never produces any output. Thus it is the dual from a security point of view to the dummy functionality and due to its unresponsiveness it is equally meaningless as an idealization of a cryptographic task. Useful functionalities will lie somewhere in between these two extremes.

This completes the description of the algebraic structure of the class of canonical functionalities. The lattice of canonical functionalities for a task can be represented by a directed graph where the $\mathcal{F}_T^{\text{top}}$ is placed at the top level and $\mathcal{F}_T^{\text{dum}}$ at the bottom. An example of such a lattice for the commitment task is given in Figure 3.

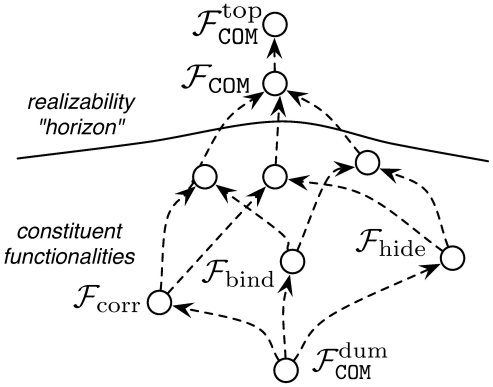

Figure 3: The lattice of canonical functionalities for commitment showing relations between the constituent functionalities. Functionalities below the "realizability horizon" can be realized in the plain model (Remark 5.28).

Given the lattice of canonical functionalities we will show that we can identify a level of the lattice as the "realizability horizon." As in [CLOS02, page 15] our notion of realizability refers only to non-trivial protocols, i.e., protocols that always generate output in the absense of adversarial interference (since a trivial protocol can realize any functionality including the $\mathcal{F}_T^{\text{top}}$ for any T). We will show that all canonical functionalities at and below this level are realizable (in the plain model) whereas all canonical functionalities above the level are unrealizable. As seen in Figure 2 we employ an error symbol for the ideal functionality to signal an inconsistency to the environment; non-trivial protocols never output this reserved error symbol (but may use other symbols for error-reporting).

Theorem 3.6. *For every task T, there is a (non-trivial) protocol π that UC-realizes the dummy functionality $\mathcal{F}_T^{\text{dum}}$. The top functionality $\mathcal{F}_T^{\text{top}}$ can only be UC-realized by a trivial protocol.*

Next, we show that UC-realizing any point $\mathcal{F}$ of $\mathscr{F}_T$ would imply that any lattice point dominated by $\mathcal{F}$ is also UC-realizable.

Theorem 3.7. *If π UC-realizes $\mathcal{F}$, then π UC-realizes any $\mathcal{F}' \lesssim \mathcal{F}$.*

As it will become apparent, the usefulness of the lattice is in the fact that it is natural to identify individual desired properties of the task, map them to canonical functionalities in the lattice and then use the conjunction operation to derive the (local) supremum of these lattice points that will yield the final functionality for the task. In this way, an idealization of a cryptographic task can be solidly "defended" by presenting its constituent canonical functionalities.

4 Deriving Canonical Ideal Functionalities

4.1 The general approach

In this section we outline a methodology for deriving canonical ideal functionalities. Given a cryptographic task T, the first step is to identify a set of consistency (including correctness) and privacy properties:

- Consistency properties are expressed in terms of languages over the I/O alphabet Σ_T. In particular one needs to identify sets of strings over Σ_T that violate a certain consistency aspect of the underlying task. Provided that the set of strings identified is polynomial-time decidable a corresponding canonical functionality is derived by setting the $\mathsf{validate}()$ predicate to reject all strings that violate the consistency property.
- Privacy properties are expressed in terms of suppression of input values that accompany action symbols. In particular, if a certain action $\mathtt{a} = (\text{ACTION}, \mathbf{P}, x)$ is supposed to maintain the privacy of a portion x' of the input x, we define $\mathsf{suppress}(\mathtt{a})$ to be equal to x with all locations corresponding to x' substituted by "$-$".

Now, using the above guidelines one can define a set of canonical functionalities each one corresponding to different security or consistency aspects of a cryptographic task. Then, given the canonical functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_k$ so defined, one can derive the canonical ideal functionality of the cryptographic task by combining the functionalities as $\mathcal{F} = \mathcal{F}_1 \wedge \ldots \wedge \mathcal{F}_k$. In such case we call $\mathcal{F}_1, \ldots, \mathcal{F}_k$ the *constituent* canonical functionalities of $\mathcal{F}$ (note that typically there will be a unique set of natural constituent functionalities although the functionality may have many different sets of possible constituents). It follows that $\mathcal{F}_i \lesssim \mathcal{F}$ and, based on Theorem 3.7, we have that any protocol that realizes $\mathcal{F}$ also realizes $\mathcal{F}_i$ for all $i = 1, \ldots, k$, thus the canonical ideal functionality $\mathcal{F}$ preserves all consistency and privacy properties identified individually in $\mathcal{F}_1, \ldots, \mathcal{F}_k$.

As an example of applying this general approach for deriving the ideal functionality of a cryptographic task the reader is referred to Section 5.2 where we apply this strategy to the case of oblivious transfer.

Depending on the cryptographic task, it may not always be easy to properly identify the required set of consistency and privacy properties that will yield the constituent canonical functionalities of the task. Fortunately, for many of them substantial effort has been spent in identifying individual security properties formalized in terms of "security games." Examples include the unforgeability game of digital signatures [GMR88] and IND-CPA game of public key encryption [GM84].

Next we show how one can leverage on existing game-based definitions of a cryptographic task to derive constituent canonical ideal functionalities in a systematic way. Importantly, our formal transformation approach from games to functionalities provably maintains the underlying game-based security notions. This translation methodology can be applied whenever game-based definitions have been identified. In fact, it is even possible to specify games for desired security properties "on demand" and apply the translation methodology to them as well.

4.2 Ideal functionalities from game-based security definitions

Individual correctness and privacy definitions are frequently specified by a game between the attacker and a "challenger" who controls different aspects of the cryptographic task. The attacker either tries to produce an undesired sequence of actions or attempts to deduce a hidden bit selected by the challenger. In the former case we call the interaction a *consistency game* while in the latter we call the interaction a *hiding game*. Examples of properties modeled with consistency games include completeness properties, the unforgeability of digital signatures, the binding property of commitments, the soundness property of zero-knowledge protocols etc., while hiding games are used to model the IND-CPA property of public-key encryption and the hiding property of commitment schemes among others. In order to detail our transformation we first provide a formal description of game based definitions.

A game-based definition G for a cryptographic task T involves two PPT interactive Turing machines, the challenger C and the attacker A. The challenger uses the actions of the cryptographic task as oracles. When the interaction terminates, a Turing machine called the judge[4] J reads the transcript of the interaction as well as the internal state of the challenger and decides which party won the game. We denote the success probability of the attacker when playing the game G by $\mathsf{Succ}_{\mathsf{A}}^{G}$; it equals the probability of the event that the judge decides that the attacker wins the game.

Consistency games are restricted to be deterministic programs (note that the task actions invoked by the game may be probabilistic). We say that a cryptographic scheme that implements a task T satisfies the property defined by a game G if for all PPT attackers A it holds that $\mathsf{Succ}_{\mathsf{A}}^{G}$ is a negligible function in the security parameter λ.

In a *hiding game*, the attacker focuses on a particular action of the cryptographic task. At some point of the interaction with the challenger, the attacker provides two input strings x_0, x_1 for a certain action where $x_0 = \langle x_0^L, x_0^R \rangle$, $x_1 = \langle x_1^L, x_1^R \rangle$ such that either the left or the right parts of the strings are required to be different while the other parts are required to be equal (for example in the witness hiding game for zero-knowledge, $x_0^L = x_1^L$ will be the statement while x_0^R, x_1^R will be two distinct witnesses). In response, the challenger flips a coin b and executes the action that is attacked on input x_b. The interaction provides the output of the action to the attacker who is supposed to provide a guess b^* for b. The judge decides that the attacker wins whenever $b = b^*$. We say that a cryptographic scheme that implements a task T satisfies the property defined by the hiding game G if for all PPT attackers A it holds that the function $|\mathsf{Succ}_{\mathsf{A}}^{G} - \frac{1}{2}|$ is a negligible function in λ.

4.2.1 Ideal functionalities from consistency games

Suppose that G is a consistency game for a cryptographic task T that involves a challenger C, an attacker A and a judge J. Let Σ be any cryptographic scheme that implements the task T. Recall that our goal is to obtain a canonical functionality $\mathcal{F}_G \in \mathscr{F}_T$ such that if a protocol π_Σ UC-realizes any $\mathcal{F} \gtrsim \mathcal{F}_G$ then the cryptographic scheme Σ satisfies the property defined by the game G. Our methodology proceeds in three steps: we first define an environment (and also the corresponding ideal world) based on the game G. Second, based on this environment, we define a language that corresponds to the event where the attacker wins the game. Third, provided that the language is decidable, we obtain a canonical functionality by incorporating the language decider as part of the functionality's $\mathsf{validate}()$ predicate. We describe the three steps in more detail below.

Step 1: Defining the environment and simulator. We first present the transformation from the game G for a task T implemented by a scheme Σ to the corresponding environment $\mathcal{Z}_G^{\mathsf{A}}$ and the ideal world adversary $\mathcal{S}_G^{\Sigma}$. We say that the transformation is sound, provided that the judge J decides that the attacker wins the game if and only if $\mathcal{Z}_G^{\mathsf{A}}$ returns 1 in an execution with $\mathcal{F}_T^{\mathrm{dum}}$ and $\mathcal{S}_G^{\Sigma}$. More specifically, it holds that $\Pr[\mathrm{IDEAL}_{\mathcal{F}_T^{\mathrm{dum}}, \mathcal{Z}_G^{\mathsf{A}}, \mathcal{S}_G^{\Sigma}}(1^\lambda) = 1] = \mathsf{Succ}_{\mathsf{A}}^{G}$.

First, we describe how we derive the environment $\mathcal{Z}_G^{\mathsf{A}}$ based on the game G. $\mathcal{Z}_G^{\mathsf{A}}$ will simulate both the attacker A and the challenger C; whenever C makes an oracle call to some action of the task, the environment $\mathcal{Z}_G^{\mathsf{A}}$ issues the corresponding ACTION symbol. The program of C will be executed by $\mathcal{Z}_G^{\mathsf{A}}$. For example, in the unforgeability game for digital signatures, an oracle call to the key generation operation will result in issuing the symbol $(\textsc{KeyGen}, \langle S, sid \rangle)$ to a party called S, where S is a random name from the namespace for some random sid; subsequent calls by C to the signing oracle for a message m, will result in the symbols $(\textsc{Sign}, \langle S, sid \rangle, m)$ directed to the same party S. Continuing with the description of $\mathcal{Z}_G^{\mathsf{A}}$, if the attacker A needs to play the role of some party of the cryptographic task T, $\mathcal{Z}_G^{\mathsf{A}}$ will need to spawn a party and corrupt it and then simulate it according to the operation of A.

Second, we define an ideal world adversary $\mathcal{S}_G^{\Sigma}$ that will be paired with $\mathcal{Z}_G^{\mathsf{A}}$. $\mathcal{S}_G^{\Sigma}$ will interact with $\mathcal{Z}_G^{\mathsf{A}}$ to corrupt parties if the environment requests it and it will also provide influence action symbols whenever a leak action symbol occurs following the program of the scheme Σ.

[4] Typically, the functionality of the judge is incorporated as part of the challenger program; we find it more convenient to specify it as a separate function.

Step 2: Defining the "bad language." This language will correspond to the event that the attacker wins the game. It is denoted by $B_{T,G} \subseteq \bigcup_{\mathsf{A},\Sigma} L_{\mathcal{F}_T^{\mathrm{dum}},\mathcal{Z}_G^{\mathsf{A}},\mathcal{S}_G^{\Sigma}}$ and contains those strings for which $\mathcal{Z}_G^{\mathsf{A}}$ returns 1. It is easy to see that given the way the transformation of the game G to the environment $\mathcal{Z}_G^{\mathsf{A}}$ was performed, those strings exactly correspond to the event when the attacker A wins the game G against the challenger C.

Now, while this language captures the event that the attacker wins the game, it is not sufficient for describing the winning event within more complex executions because the bad sequence of symbols may be interleaved with other actions. This calls for the specification of an *extended* bad language that would contain all possible bad symbol sequences. It is tempting to define such language as the set of all strings that have a substring in $B_{T,G}$. However, while this works in many cases it would fail for any property where it is possible for good strings to be superstrings of bad ones (for example, in the case of signatures, a single valid VERIFY symbol is a bad string while if the same symbol is preceded by a corresponding SIGN symbol is a good string). To address this monotonicity issue a point of reference in the form of a (restricted) good language would be helpful.

It turns out that this role can be played successfully by the "attack-fail" language, denoted by $A_{T,G} \subseteq \bigcup_{\mathsf{A},\Sigma} L_{\mathcal{F}_T^{\mathrm{dum}},\mathcal{Z}_G^{\mathsf{A}},\mathcal{S}_G^{\Sigma}}$, that we define as the set of all strings that the environment $\mathcal{Z}_G^{\mathsf{A}}$ returns 0; in other words, these are strings that structurally resemble the strings in the bad language but where the judge predicate proclaims the challenger as the winner. Armed with these notions, we now define the *extended* bad language to be those strings w of $L_{\mathcal{F}_T^{\mathrm{dum}}}$ that contain as a subsequence a string w' in language $B_{T,G}$ where w' has no supersequence in language $A_{T,G}$ (which in turn is a subsequence of w). Formally, we write this language as

$$B_{T,G}^{\mathrm{ext}} = \left\{ w \;\middle|\; \exists w' \preccurlyeq w \text{ s.t. } w' \in B_{T,G} \wedge w' \notin A_{T,G}^{\preccurlyeq} \cap \{w\}^{\preccurlyeq} \right\}.$$

Step 3: Defining the ideal functionality. In order to define the class of canonical functionalities that capture the game G we need first to show that the extended bad language $B_{T,G}^{\mathrm{ext}}$ defined in step 2 is polynomial-time decidable. Then, given the decider D for the language, we define the canonical functionality $\mathcal{F}_G$ that captures the game G by requiring that $\mathsf{validate}(w) = 0$ if and only if $w \in B_{T,G}^{\mathrm{ext}}$; in other words, the function $\mathsf{validate}()$ simulates the decider D, and whenever the decider accepts the functionality halts.

We now show that the translation detailed above is sound.

Theorem 4.1. *Assume that a scheme Σ implements a task T and G is a consistency game for T. It holds that if π_Σ UC-realizes some $\mathcal{F} \gtrsim \mathcal{F}_G$ then Σ satisfies the property defined by G.*

Remark 4.2. While investigating the opposite direction of the above theorem for arbitrary consistency games (namely, if a given scheme satisfies a property, then the corresponding protocol UC-realizes the derived functionality) is beyond the scope of this paper, we note that for the consistency games studied in this paper (i.e., unforgeability, consistency, and completeness games for digital signatures), we show that the opposite direction also holds.

4.2.2 Ideal functionalities from hiding games

Let G be a hiding game for a cryptographic task T. We show how to define a canonical functionality for the task that implies the hiding property. In this case, our methodology proceeds in two steps: we first define an environment and an ideal world simulator based on the game G. Second, based on the environment's operation we define the canonical functionality by appropriately modifying the $\mathsf{suppress}$ function.

Step 1: Defining the environment and simulator. As in the case of consistency games, we define an environment $\mathcal{Z}_G^{\mathsf{A}}$ and simulator $\mathcal{S}_G^{\Sigma}$ based on the operation of the challenger C, the attacker A, the judge J and the scheme Σ. The transformation is identical to the one in step 1 in Section 4.2.1.

Step 2: Defining the canonical functionality. During any execution of the environment $\mathcal{Z}_G^{\mathsf{A}}$ with $\mathcal{S}_G^{\Sigma}$, it holds that the environment issues an ACTION symbol with input x_b where b is a random bit selected by $\mathcal{Z}_G^{\mathsf{A}}$ and x_0, x_1 were provided by the attacker A (which is simulated by $\mathcal{Z}_G^{\mathsf{A}}$). Assuming that $x_0 = \langle x_0^L, x_0^R \rangle$ and $x_1 = \langle x_1^L, x_1^R \rangle$ and the game G contains the test $x_0^L = x_1^L$ and $x_0^R \neq x_1^R$, we define the $\mathsf{suppress}$ function for symbol $\mathsf{a} = (\text{ACTION}, \mathbf{P}, x_b)$ where $b \in \{0,1\}$ by $\mathsf{suppress}(\mathsf{a}) = \langle x_b^L, (-)^{|x_b^R|} \rangle$ (recall that $\mathsf{suppress}(\mathsf{a}) \prec x_b$).

Theorem 4.3. *Suppose that a cryptographic scheme Σ implements a cryptographic task T and G is a hiding game for T. Then it holds that if π_Σ UC-realizes some $\mathcal{F} \gtrsim \mathcal{F}_G$, then Σ satisfies the hiding property defined by game G.*

5 Applying the Methodology

We now apply our methodology to a variety of cryptographic tasks, including digital signatures, oblivious transfer, commitments and ZK proofs of knowledge. In some cases the canonical functionalities we derive are equivalent to the ones that have been identified before, where in others their differences point to subtle definitional problems in their previous formulations (despite successive revisions).

5.1 Digital signatures

The basic requirements for digital signatures, completeness, consistency and unforgeability, were first formulated in [GMR88][5]. Each property is specified by a consistency game. In this section we show how to translate these traditional notions into the corresponding canonical functionalities, from which we obtain $\mathcal{F}_{\mathtt{SIG}}$, and conclude with a comparison to previous signature functionalities.

Following Figure 2, any canonical signature functionality $\mathcal{F}_{\mathtt{SIG}}$ is defined for two types of roles, the signer S and the verifier V, with three actions, KEYGEN, SIGN, VERIFY. We denote the canonical signature functionality class as $\mathscr{F}_{\mathtt{SIG}}$.

5.1.1 Unforgeability

Definition 5.1 (Unforgeability [GMR88])**.** *A signature scheme* $\Sigma(\mathtt{SIG}) = \langle \mathtt{gen}, \mathtt{sign}, \mathtt{verify} \rangle$ *is* unforgeable *if for all PPT attackers* A*,*

$$\Pr\left[\begin{array}{r}(vk, sk) \leftarrow \mathtt{gen}(1^\lambda); (m, \sigma) \leftarrow \mathsf{A}^{\mathtt{sign}(vk,sk,\cdot)}(vk); \phi \leftarrow \mathtt{verify}(vk, m, \sigma) \\ : \phi = 1 \textit{ and } \mathsf{A} \textit{ never submitted } m \textit{ to the } \mathtt{sign}(vk, sk, \cdot) \textit{ oracle}\end{array}\right] \leq \mathsf{negl}(\lambda)$$

The above definition can be formulated as a consistency game G_{uf} for the task SIG as follows: the challenger C uses algorithms $\mathtt{gen}(), \mathtt{sign}(), \mathtt{verify}()$ as oracles, and interacts with the attacker A: the challenger C queries the $\mathtt{gen}()$ oracle and obtains $\langle sk, vk \rangle$, and then sends such vk to A; each time upon receiving m from the attacker A, the challenger C queries the $\mathtt{sign}()$ oracle with m and obtains σ, and then returns σ to A; upon receiving from A a pair $\langle m', \sigma' \rangle$, C queries the $\mathtt{verify}()$ oracle with $\langle m', \sigma', vk \rangle$ and obtains the verification result. The judge J decides that A wins the game if m' has never been queried before and the verification result is 1.

Step 1. Based on the game G_{uf} described above, we can construct an environment $\mathcal{Z}_{\mathrm{uf}}^{\mathsf{A}}$ and the corresponding ideal world adversary $\mathcal{S}_{\mathrm{uf}}^{\Sigma}$ as follows. In order to simulate the game, the environment first picks S and V from the namespace at random as well as a random sid. The environment sends $(\text{KEYGEN}, \langle S, sid \rangle)$ to party S and receives $(\text{KEYGENRETURN}, \langle S, sid \rangle, vk)$; then the environment simulates A on input vk; when A queries m to its signing oracle, the environment sends $(\text{SIGN}, \langle S, sid \rangle, m)$ to party S and returns the output of S to A. Once A outputs a pair $\langle m, \sigma \rangle$, the environment sends $(\text{VERIFYRETURN}, \langle V, sid \rangle, \langle m, \sigma, vk \rangle)$ to some party V and receives the verification result ϕ. In the case that m has never been queried and $\phi = 1$, the environment terminates with 1; otherwise with 0.

We next define the ideal-world adversary $\mathcal{S}_{\mathrm{uf}}^{\Sigma}$. Each time $\mathcal{S}_{\mathrm{uf}}^{\Sigma}$ receives $(\text{LEAKKEYGEN}, \langle S, sid \rangle)$ from the ideal functionality, it runs $(vk, sk) \leftarrow \mathtt{gen}(1^\lambda)$ and sends $(\text{INFLKEYGEN}, \langle S, sid \rangle, vk)$ to the functionality. When $\mathcal{S}_{\mathrm{uf}}^{\Sigma}$ receives $(\text{LEAKSIGN}, \langle S, sid \rangle, m)$ from the ideal functionality, it runs $\sigma \leftarrow \mathtt{sign}(vk, sk, m)$, and sends $(\text{INFLSIGN}, \langle S, sid \rangle, \sigma)$ to the functionality. When $\mathcal{S}_{\mathrm{uf}}^{\Sigma}$ receives $(\text{LEAKVERIFY}, \langle V, sid \rangle, \langle m, \sigma, vk \rangle)$ from the ideal functionality, it runs $\phi \leftarrow \mathtt{verify}(vk, sk, m, \sigma)$, and sends $(\text{INFLVERIFY}, \langle V, sid \rangle, \phi)$ to the functionality.

[5] Consistency is implied in the GMR specification, as pointed out by Canetti [Can04].

Step 2. For any adversary A and signature scheme Σ we define $L_{\mathcal{F}_{\mathtt{SIG}}^{\mathrm{dum}},\mathcal{Z}_{\mathrm{uf}}^{\mathsf{A}},\mathcal{S}_{\mathrm{uf}}^{\Sigma}}$ (cf. Section 3.2) with $\mathcal{Z}_{\mathrm{uf}}^{\mathsf{A}}, \mathcal{S}_{\mathrm{uf}}^{\Sigma}$ as defined in step 1. We next define the set of strings $B_{\mathtt{SIG},\mathrm{uf}}$ as the subset of $\bigcup_{\mathsf{A},\Sigma} L_{\mathcal{F}_{\mathtt{SIG}}^{\mathrm{dum}},\mathcal{Z}_{\mathrm{uf}}^{\mathsf{A}},\mathcal{S}_{\mathrm{uf}}^{\Sigma}}$ that contains exactly those strings for which the environment returns 1.

Lemma 5.2. *(1)* $B_{\mathtt{SIG},\mathrm{uf}} = \left\{ w \;\middle|\; \begin{array}{l} w = (\textsc{KeyGen}, \langle S, sid\rangle) \\ \quad (\textsc{KeyGenReturn}, \langle S, sid\rangle, vk) \\ \quad (\textsc{Sign}, \langle S, sid\rangle, m_1) \\ \quad (\textsc{SignReturn}, \langle S, sid\rangle, \sigma_1) \\ \quad \ldots \\ \quad (\textsc{Sign}, \langle S, sid\rangle, m_\ell) \\ \quad (\textsc{SignReturn}, \langle S, sid\rangle, \sigma_\ell) \\ \quad (\textsc{Verify}, \langle V, sid\rangle, \langle m', \sigma', vk\rangle) \\ \quad (\textsc{VerifyReturn}, \langle V, sid\rangle, \phi) \\ \textit{such that} \\ \phi = 1 \wedge m' \notin \{m_1, \ldots, m_\ell\} \end{array} \right\}$,
and (2) $B_{\mathtt{SIG},\mathrm{uf}}$ *is decidable in polynomial time.*

Following the methodology in Section 4.2, by changing the judge decision condition into its negation, i.e., $\phi = 0 \vee m' \in \{m_1, \ldots, m_\ell\}$, we can define the "attack-fail" language $A_{\mathtt{SIG},\mathrm{uf}}$ (in fact in this way we define a strict subset of the attack-fail language but this is sufficient for our purposes; for the complete specification of this language refer to Appendix A). We further define the extended bad language $B_{\mathtt{SIG},\mathrm{uf}}^{\mathrm{ext}}$ as indicated there, and observe that $B_{\mathtt{SIG},\mathrm{uf}}^{\mathrm{ext}}$ is also decidable in polynomial time.

Step 3. Next we define the class of ideal functionalities that corresponds to the unforgeability property.

Definition 5.3 (Canonical Functionality $\mathcal{F}_{\mathrm{uf}}$)**.** *The functionality* $\mathcal{F}_{\mathrm{uf}} \in \mathscr{F}_{\mathtt{SIG}}$ *equals* $\mathcal{F}_{\mathtt{SIG}}^{\mathsf{suppress},\mathsf{validate}}$*, where (1)* $\mathsf{suppress}()$ *satisfies that for all* x *and any* $\textsc{Action} \in \{\textsc{KeyGen}, \textsc{Sign}, \textsc{Verify}\}$, $\mathsf{suppress}((\textsc{Action}, \mathbf{P}, x)) = x$*, (i.e., the same as in* $\mathcal{F}_{\mathtt{SIG}}^{\mathrm{dum}}$*), and (2)* $\mathsf{validate}(w) = 0$ *if and only if* $w \in B_{\mathtt{SIG},\mathrm{uf}}^{\mathrm{ext}}$.

Based on Theorem 4.1, we have the following corollary:

Corollary 5.4. *If* $\pi_{\Sigma(\mathtt{SIG})}$ *realizes some* $\mathcal{F} \gtrsim \mathcal{F}_{\mathrm{uf}}$*, then* $\Sigma(\mathtt{SIG})$ *is unforgeable.*

Further, we show that for unforgeability the other direction also holds – i.e., the transformation is tight.

Theorem 5.5. *If* $\Sigma(\mathtt{SIG})$ *is unforgeable, then* $\pi_{\Sigma(\mathtt{SIG})}$ *realizes* $\mathcal{F}_{\mathrm{uf}}$.

5.1.2 Completeness

Informally, completeness means that the verification value of an honestly generated message-signature pair should be true except with negligible probability.

Definition 5.6 (Completeness)**.** *A signature scheme* $\Sigma(\mathtt{SIG}) = \langle \mathtt{gen}, \mathtt{sign}, \mathtt{verify}\rangle$ *is* complete *if for all PPT attackers* A*,*

$$\Pr\left[\; m \leftarrow \mathsf{A}(1^\lambda); (vk, sk) \leftarrow \mathtt{gen}(1^\lambda); \sigma \leftarrow \mathtt{sign}(vk, sk, m); \phi \leftarrow \mathtt{verify}(vk, m, \sigma) \;:\; \phi = 0 \;\right] \leq \mathsf{negl}(\lambda).$$

The above definition can be modeled as a consistency game, G_{comp} as follows. The challenger C uses algorithms $\mathtt{gen}(), \mathtt{sign}(), \mathtt{verify}()$ as oracles, and interacts with completeness attacker A: after receiving m produced by A, the challenger C queries the $\mathtt{gen}()$ oracle and obtains sk, vk; then C queries the $\mathtt{sign}()$ oracle with sk, m and obtains σ; later C queries the $\mathtt{verify}()$ oracle with $\langle m, \sigma, vk\rangle$ to obtains the verification result. The judge J decides that A wins the game if the verification result is 0.

Step 1. Based on the game G_{comp} described above, we can construct an environment $\mathcal{Z}_{\mathrm{comp}}^{\mathsf{A}}$ and the corresponding ideal world adversary $\mathcal{S}_{\mathrm{comp}}^{\Sigma}$. The environment $\mathcal{Z}_{\mathrm{comp}}^{\mathsf{A}}$ here is similar to the environment $\mathcal{Z}_{\mathrm{uf}}^{\mathsf{A}}$; the environment first

picks S and V from the namespace at random as well as a random sid. The environment simulates A with input 1^λ and obtains m; it then sends $(\text{KEYGEN}, \langle S, sid\rangle)$ to party S and receives $(\text{KEYGENRETURN}, \langle S, sid\rangle, vk)$ from the party S; later the environment sends $(\text{SIGN}, \langle S, sid\rangle, m)$ to party S and receives σ; the environment inputs $(\text{VERIFYRETURN}, \langle V, sid\rangle, \langle m, \sigma, vk\rangle)$ to V and receives the verification result. If the verification result $\phi = 0$, the environment terminates with 1; otherwise with 0. The adversary $\mathcal{S}^{\Sigma}_{\text{comp}}$ is defined similarly to the adversary $\mathcal{S}^{\Sigma}_{\text{uf}}$ in the previous section.

Step 2. For any completeness attacker A and scheme Σ, the environment $\mathcal{Z}^{\mathsf{A}}_{\text{comp}}$, the adversary $\mathcal{S}^{\Sigma}_{\text{comp}}$, and the dummy canonical signature functionality together give rise to the language $L_{\mathcal{F}^{\text{dum}}_{\mathtt{SIG}}, \mathcal{Z}^{\mathsf{A}}_{\text{comp}}, \mathcal{S}^{\Sigma}_{\text{comp}}}$. We consider the subset of strings $B_{\mathtt{SIG},\text{comp}}$ of the union of all the languages quantified over all possible completeness attackers A and schemes Σ that contains exactly those strings for which the environment returns 1. Formally, $B_{\mathtt{SIG},\text{comp}} \subseteq \bigcup_{\mathsf{A},\Sigma} L_{\mathcal{F}^{\text{dum}}_{\mathtt{SIG}}, \mathcal{Z}^{\mathsf{A}}_{\text{comp}}, \mathcal{S}^{\Sigma}_{\text{comp}}}$.

We next prove the following characterization of this language as well as determine its time complexity:

Lemma 5.7. *(1)* $B_{\mathtt{SIG},\text{comp}} = \left\{ w \;\middle|\; \begin{array}{l} w = (\text{KEYGEN}, \langle S, sid\rangle) \\ \quad (\text{KEYGENRETURN}, \langle S, sid\rangle, vk) \\ \quad (\text{SIGN}, \langle S, sid\rangle, m) \\ \quad (\text{SIGNRETURN}, \langle S, sid\rangle, \sigma) \\ \quad (\text{VERIFY}, \langle V, sid\rangle, \langle m, \sigma, vk\rangle) \\ \quad (\text{VERIFYRETURN}, \langle V, sid\rangle, \phi) \\ \textit{such that } \phi = 0 \end{array} \right\}$,
(2) $B_{\mathtt{SIG},\text{comp}}$ *is decidable in polynomial time.*

Following the methodology in Section 4.2, by changing the judge decision condition $\phi = 0$ in $B_{\mathtt{SIG},\text{comp}}$ into its negation, i.e., $\phi = 1$, we can similarly define the attack-fail language $A_{\mathtt{SIG},\text{comp}}$. Note that $B_{\mathtt{SIG},\text{comp}}$ and $A^{\preccurlyeq}_{\mathtt{SIG},\text{comp}}$ are disjoint. We observe that $B^{\text{ext}}_{\mathtt{SIG},\text{comp}}$ is also decidable in polynomial time.

Step 3. We now define the class of ideal functionalities that corresponds to the completeness property.

Definition 5.8 (Canonical Functionality $\mathcal{F}_{\text{comp}}$)**.** *The functionality* $\mathcal{F}_{\text{comp}} \in \mathscr{F}_{\mathtt{SIG}}$ *equals* $\mathcal{F}^{\mathsf{suppress},\mathsf{validate}}_{\mathtt{SIG}}$*, where (1)* $\mathsf{suppress}()$ *is the same as in* $\mathcal{F}^{\text{dum}}_{\mathtt{SIG}}$*, and (2)* $\mathsf{validate}(w) = 0$ *if and only if* $w \in B^{\text{ext}}_{\mathtt{SIG},\text{comp}}$.

The following corollary follows from Theorem 4.1.

Corollary 5.9. *If* $\pi_{\Sigma(\mathtt{SIG})}$ *realizes some* $\mathcal{F} \gtrsim \mathcal{F}_{\text{comp}}$*, then* $\Sigma(\mathtt{SIG})$ *is complete.*

The other direction also holds in this case:

Theorem 5.10. *If* $\Sigma(\mathtt{SIG})$ *is complete, then* $\pi_{\Sigma(\mathtt{SIG})}$ *realizes* $\mathcal{F}_{\text{comp}}$.

5.1.3 Consistency

Informally, consistency of a signature scheme means that verifying the same message-signature pair twice will return two different verification values with only negligible probability. More formally:

Definition 5.11 (Consistency)**.** *A signature scheme* $\Sigma(\mathtt{SIG}) = \langle \mathtt{gen}, \mathtt{sign}, \mathtt{verify}\rangle$ *is* consistent *if for all PPT attackers* A*,*

$$\Pr\left[\ (vk, m, \sigma) \leftarrow \mathsf{A}(1^\lambda); \phi_1 \leftarrow \mathtt{verify}(vk, m, \sigma); \phi_2 \leftarrow \mathtt{verify}(vk, m, \sigma) \ :\ \phi_1 \neq \phi_2 \ \right] \leq \mathsf{negl}(\lambda).$$

The above definition can also be modeled by a consistency game, G_{cons}, as follows. The challenger C uses algorithms $\mathtt{gen}(), \mathtt{sign}(), \mathtt{verify}()$ as oracles, and interacts with the consistency attacker A: C simulates A on input 1^λ to obtain $\langle vk, m, \sigma\rangle$ and then calls the the $\mathtt{verify}()$ oracle with $\langle m, \sigma, vk\rangle$ twice and obtains the verification results ϕ_1 and ϕ_2 respectively. The judge J decides that A wins the game if the two verification results are different, i.e., $\phi_1 \neq \phi_2$.

Step 1. Based on the game G_{cons} described above, we can construct an environment $\mathcal{Z}^{\mathsf{A}}_{\text{cons}}$ and the corresponding ideal world adversary $\mathcal{S}^{\Sigma}_{\text{cons}}$ as follows. The environment first picks S and two V's from the namespace at random as well as a random sid. Then the environment simulates A to obtain $\langle vk, m, \sigma\rangle$ and gives the symbols $(\text{VERIFY}, \langle V_1, sid\rangle, \langle m, \sigma, vk\rangle)$ and $(\text{VERIFY}, \langle V_2, sid\rangle, \langle m, \sigma, vk\rangle)$ to obtain the symbols $(\text{VERIFYRETURN}, \langle V_1, sid\rangle, \phi_1)$ and $(\text{VERIFYRETURN}, \langle V_2, sid\rangle, \phi_2)$. In the case that $\phi_1 \neq \phi_2$, the environment terminates with 1 otherwise with 0. $\mathcal{S}^{\Sigma}_{\text{cons}}$ is defined similarly to $\mathcal{S}^{\Sigma}_{\text{uf}}$.

Step 2. For any consistency attacker A and scheme Σ, the environment $\mathcal{Z}^{\mathsf{A}}_{\text{cons}}$, the ideal adversary $\mathcal{S}^{\Sigma}_{\text{cons}}$, and the dummy canonical signature functionality together give rise to the language $L_{\mathcal{F}^{\text{dum}}_{\mathtt{SIG}}, \mathcal{Z}^{\mathsf{A}}_{\text{cons}}, \mathcal{S}^{\Sigma}_{\text{cons}}}$. We consider the subset of strings $B_{\mathtt{SIG},\text{cons}}$ of the union of all the languages quantified over all possible consistency attackers A and schemes Σ that contains exactly those strings for which the environment returns 1. Formally, $B_{\mathtt{SIG},\text{cons}} \subseteq \bigcup_{\mathsf{A},\Sigma} L_{\mathcal{F}^{\text{dum}}_{\mathtt{SIG}}, \mathcal{Z}^{\mathsf{A}}_{\text{cons}}, \mathcal{S}^{\Sigma}_{\text{cons}}}$. We next prove the following characterization of this language as well as determine its time complexity:

Lemma 5.12. *(1)* $B_{\mathtt{SIG},\text{cons}} = \left\{ w \;\middle|\; \begin{array}{l} w = (\text{VERIFY}, \langle V_1, sid\rangle, \langle m, \sigma, vk\rangle) \\ \quad\; (\text{VERIFYRETURN}, \langle V_1, sid\rangle, \phi_1) \\ \quad\; (\text{VERIFY}, \langle V_2, sid\rangle, \langle m, \sigma, vk\rangle) \\ \quad\; (\text{VERIFYRETURN}, \langle V_2, sid\rangle, \phi_2) \\ \textit{such that } \phi_1 \neq \phi_2 \end{array} \right\}$,

and (2) $B_{\mathtt{SIG},\text{cons}}$ *is decidable in polynomial time.*

Following the methodology in Section 4.2, by changing the judge decision condition $\phi_1 \neq \phi_2$ in $B_{\mathtt{SIG},\text{cons}}$ into its negation, i.e., $\phi_1 = \phi_2$, we can similarly define the attack-fail language $A_{\mathtt{SIG},\text{cons}}$. Note that $B_{\mathtt{SIG},\text{cons}}$ and $A^{\preccurlyeq}_{\mathtt{SIG},\text{cons}}$ are disjoint. We observe that $B^{\text{ext}}_{\mathtt{SIG},\text{cons}}$ is also decidable in polynomial time.

Step 3. We proceed next to define the canonical functionality that corresponds to the consistency property.

Definition 5.13 (Canonical Functionality $\mathcal{F}_{\text{cons}}$)**.** *The functionality* $\mathcal{F}_{\text{cons}} \in \mathscr{F}_{\mathtt{SIG}}$ *equals* $\mathcal{F}^{\mathsf{suppress},\mathsf{validate}}_{\mathtt{SIG}}$*, where (1)* $\mathsf{suppress}()$ *is the same as in* $\mathcal{F}^{\text{dum}}_{\mathtt{SIG}}$*, and (2)* $\mathsf{validate}(w) = 0$ *if and only if* $w \in B^{\text{ext}}_{\mathtt{SIG},\text{cons}}$.

The following corollary also follows from Theorem 4.1:

Corollary 5.14. *If* $\pi_{\Sigma(\mathtt{SIG})}$ *realizes some* $\mathcal{F} \gtrsim \mathcal{F}_{\text{cons}}$*, then* $\Sigma(\mathtt{SIG})$ *is consistent.*

In the case of consistency, the other direction also holds:

Theorem 5.15. *If* $\Sigma(\mathtt{SIG})$ *is consistent, then* $\pi_{\Sigma(\mathtt{SIG})}$ *realizes* $\mathcal{F}_{\text{cons}}$.

Remark 5.16. We first recall the definition of consistency as given in [Can04]. We call it "weak consistency" as it restricts the adversary by requiring honest key generation.

Definition 5.17. *A scheme* $\Sigma(\mathtt{SIG}) = \langle \mathtt{gen}, \mathtt{sign}, \mathtt{verify}\rangle$ *is* weakly consistent *if for all PPT attackers* A*,*

$$\Pr\left[\begin{array}{l} (vk, sk) \leftarrow \mathtt{gen}(1^\lambda); (m, \sigma) \leftarrow \mathsf{A}^{\mathtt{sign}(vk, sk, \cdot)}(vk); \\ \phi_1 \leftarrow \mathtt{verify}(vk, m, \sigma); \phi_2 \leftarrow \mathtt{verify}(vk, m, \sigma) \;:\; \phi_1 \neq \phi_2 \end{array}\right] \leq \mathsf{negl}(\lambda).$$

We now construct a counterexample Σ' which satisfies completeness, unforgeability and weak consistency as defined above, but in which the corresponding $\pi_{\Sigma'}$ does not realize $\mathcal{F}_{\mathtt{SIG}}$ in [Can04] (or the $\mathcal{F}_{\mathtt{SIG}}$ that is produced from our translation methodology).

Let Σ be a scheme that satisfies completeness, unforgeability and weak consistency. We modify such Σ into Σ': (1) prepend a bit b to the verification key; if $b = 0$ then the verification procedure remains the same; if $b = 1$ then the verification procedure accepts its input message-signature pair with probability $1/2$; (2) the key generation algorithm returns a verification key starting with bit 0. Notice that Σ' still satisfies the three properties, completeness, unforgeability and weak consistency, since the honest key generation will never return a verification key starting with bit 1. According to Theorem 2 in [Can04], the corresponding $\pi_{\Sigma'}$ would realize $\mathcal{F}_{\mathtt{SIG}}$. This, however, does not hold. When the signer is corrupted at the beginning of the execution, a verification key vk' with starting bit 1 can be chosen and then two verification requests with the same input $\langle m, \sigma, vk'\rangle$ will return different verification results with non-negligible probability — $1/2$ in this case.

5.1.4 The canonical ideal signature functionality

The (canonical) ideal signature functionality $\mathcal{F}_{\mathtt{SIG}} = \mathcal{F}_{\text{uf}} \wedge \mathcal{F}_{\text{comp}} \wedge \mathcal{F}_{\text{cons}}$ and is shown in Figure 4. In light of Theorem 3.7 we obtain the following:

Corollary 5.18. *If $\pi_{\Sigma(\mathtt{SIG})}$ realizes some $\mathcal{F} \gtrsim \mathcal{F}_{\text{uf}} \wedge \mathcal{F}_{\text{comp}} \wedge \mathcal{F}_{\text{cons}}$, then the signature scheme $\Sigma(\mathtt{SIG})$ satisfies the game-based properties of unforgeability, completeness, and consistency.*

Canonical Signature Functionality $\mathcal{F}_{\mathtt{SIG}}$

Actions: KEYGEN, SIGN, VERIFY

Well-formedness ($\mathsf{WF}_{\mathtt{SIG}}$): Any $(\text{SIGN}, \langle S, sid\rangle, \cdot)$ symbol must be preceded by a $(\text{KEYGEN}, \langle S, sid\rangle)$ symbol.

Public and Secret Outputs ($\mathsf{PO}_{\mathtt{SIG}}$, $\mathsf{SO}_{\mathtt{SIG}}$): For all $w, \mathtt{a}$, $\mathsf{PO}_{\mathtt{SIG}}(w, \mathtt{a}) = \epsilon$, and $\mathsf{SO}_{\mathtt{SIG}}(w, \mathtt{a}) = \epsilon$.

Suppress and Validate: (1) $\mathsf{suppress}()$ satisfies that for all x and any $\text{ACTION} \in \{\text{KEYGEN}, \text{SIGN}, \text{VERIFY}\}$, $\mathsf{suppress}((\text{ACTION}, \mathbf{P}, x)) = x$, (2) $\mathsf{validate}(w) = 1$ iff $w \notin B^{\text{ext}}_{\mathtt{SIG},\text{uf}}$ and $w \notin B^{\text{ext}}_{\mathtt{SIG},\text{comp}}$ and $w \notin B^{\text{ext}}_{\mathtt{SIG},\text{cons}}$.

Figure 4: Ideal functionality for digital signature based on the canonical functionality template.

5.1.5 Comparison to previous signature functionalities

Canonical functionality $\mathcal{F}_{\mathtt{SIG}}$ from Figure 4 can be shown to be UC-equivalent to the digital signature ideal functionality given in [Can04]. This can be done by showing that the dummy protocol in the $\mathcal{F}_{\mathtt{SIG}}$-hybrid world realizes unconditionally $\mathcal{F}^{\text{Can}}_{\mathtt{SIG}}$ as well as the dummy protocol in the $\mathcal{F}^{\text{Can}}_{\mathtt{SIG}}$-hybrid world realizes unconditionally our $\mathcal{F}_{\mathtt{SIG}}$. The proof requires the construction of two ideal world simulators, one that interacts with the ideal functionality $\mathcal{F}_{\mathtt{SIG}}$ and simulates the view of any environment operating in the $\mathcal{F}^{\text{Can}}_{\mathtt{SIG}}$ hybrid world as well as a simulator that interacts with the functionality $\mathcal{F}^{\text{Can}}_{\mathtt{SIG}}$ and simulates the view of any environment that operates in the $\mathcal{F}_{\mathtt{SIG}}$ hybrid world. The proof is similar to the proof of Proposition 5.22, where we demonstrate that our canonical oblivious transfer ideal functionality is UC equivalent to that of [CLOS02]); we refer to that section for more details.

However, our canonical functionality capturing the consistency property as described above is derived from a game-based definition for consistency that is *different* from the one in [Can04]. The reason is that the consistency formulation given there falls short of capturing the intended properties for the digital signature task in the UC setting. We elaborate on this issue below.

Recall that a first rendering of $\mathcal{F}_{\mathtt{SIG}}$ [Can01] failed to capture the consistency property, as pointed out in [BH04]. The latter work, however, did not capture consistency fully either as was in turn pointed out in [Can04], which performed a thorough investigation between the correspondence of the game-based security formulation of the Goldwasser *et al.* [GMR88] notion for digital signatures and the $\mathcal{F}_{\mathtt{SIG}}$ ideal functionality. Indeed, a correspondence theorem was shown in [Can04] establishing that any digital signature scheme secure in the GMR sense would result in a UC-secure signature protocol.

However, as we now show with the help of our methodology, this correspondence does not stand. In fact, when one applies our translation methodology to the three game-based definitions that are put forth in [Can04] to capture the [GMR88] notion of security, the resulting functionality is not the $\mathcal{F}_{\mathtt{SIG}}$ functionality as defined above. This is due to the fact that the consistency game as defined in [Can04] (cf. page 12, Definition 1) assumes an honest key generation. More specifically, if our consistency game translation is applied to that game, it results in the following bad language (see Lemma 5.12):

$$B'_{\mathtt{SIG},\text{cons}} = \left\{ w \;\middle|\; \begin{array}{l} w = (\text{KEYGEN}, \langle S, sid\rangle) \\ \quad (\text{KEYGENRETURN}, \langle S, sid\rangle, vk) \\ \quad (\text{VERIFY}, \langle V_1, sid\rangle, \langle m, \sigma, vk\rangle) \\ \quad (\text{VERIFYRETURN}, \langle V_1, sid\rangle, \phi_1) \\ \quad (\text{VERIFY}, \langle V_2, sid\rangle, \langle m, \sigma, vk\rangle) \\ \quad (\text{VERIFYRETURN}, \langle V_2, sid\rangle, \phi_2) \\ \text{such that } \phi_1 \neq \phi_2 \end{array} \right\}$$

It follows that the corresponding canonical functionality $\mathcal{F}'_{\text{cons}}$ would have a validate predicate that checks for verification inconsistency *only in the case* that a KEYGEN symbol has been recorded in the history of the functionality. This is too restrictive as it precludes corrupted signers that may never register a KEYGEN symbol with the functionality (and in fact this is exactly the issue pointed out in [Can04] regarding the previous work of [BH04]).

It is easy to see that the resulting (weaker) canonical functionality $\mathcal{F}'_{\mathtt{SIG}} = \mathcal{F}_{\text{uf}} \wedge \mathcal{F}_{\text{comp}} \wedge \mathcal{F}'_{\text{cons}}$ resides at a lower point compared to $\mathcal{F}_{\mathtt{SIG}}$ in the $\mathscr{F}_{\mathtt{SIG}}$ lattice. This is due to the fact that $B^{\text{ext}}_{\mathtt{SIG},\text{cons}} \supseteq B'^{\text{ext}}_{\mathtt{SIG},\text{cons}}$ where $B'^{\text{ext}}_{\mathtt{SIG},\text{cons}}$ is the extended bad language that corresponds to the bad language $B'_{\mathtt{SIG},\text{cons}}$. Furthermore, as shown in Remark 5.16, it is possible to design a digital signature scheme Σ' so that its corresponding protocol $\pi_{\Sigma'}$ UC-realizes $\mathcal{F}'_{\mathtt{SIG}}$ but fails to realize $\mathcal{F}_{\mathtt{SIG}}$. This scheme passes the game-based formulation in [Can04] and, based on our methodology, it will UC-realize $\mathcal{F}'_{\mathtt{SIG}}$; nonetheless, $\mathcal{F}_{\mathtt{SIG}}$ will not be realized by this digital signature. As a result, the appropriate formulation of the consistency game (from which we derive the language $B_{\mathtt{SIG},\text{cons}}$) is the one presented in Definition 5.11, and this provides the exact game-based correspondence to the $\mathcal{F}_{\mathtt{SIG}}$ canonical functionality.

Finally, as mentioned in Section 3, the current formulation of our framework does not attempt to capture the "object-oriented insulation" of functionalities. Thus, in the case of signatures, the adversary is notified of the event of a signature generation and is capable of influencing the way the signatures are represented as strings. While this formulation is more general, see the treatment of signatures and discussions in [Can05, Pat05] about applications where the locally run, "object" approach where the functionality does more than providing an authenticated message delivery service might be desired.

5.2 Oblivious transfer

We consider the 1-out-of-2 version of oblivious transfer [Rab81, EGL85, Cré87]. The $\mathcal{F}_{\mathtt{OT}}$ functionality is defined for two roles, the sender S and the receiver R. The actions, well-formedness, and public and secret outputs of $\mathcal{F}_{\mathtt{OT}}$ are given in Figure 5. We next describe the three constituent canonical functionalities of oblivious transfer that correspond to its three basic properties: correctness, sender privacy and receiver privacy.

5.2.1 Correctness

In order to obtain the bad language for correctness, we observe that for every two messages (m_0, m_1) from the sender and every selection bit i from the receiver, the value the receiver obtains should be equal to m_i. Based on this, we identify the set of strings that are inconsistent with the correctness property as:

$$B_{\mathtt{OT},\text{corr}} = \left\{ w \;\middle|\; \begin{array}{l} w = \mathtt{abc} \text{ or } \mathtt{bac} \text{ where} \\ \mathtt{a} = (\text{TRANSFER}, \langle\langle S, R, sid\rangle, S\rangle, \langle m_0, m_1\rangle), \\ \mathtt{b} = (\text{TRANSFER}, \langle\langle S, R, sid\rangle, R\rangle, i), \\ \mathtt{c} = (\text{TRANSFERRETURN}, \langle\langle S, R, sid\rangle, R\rangle, m'), \\ \text{such that } m' \neq m_i \end{array} \right\}$$

Following the methodology in Section 4.2, by changing the judge decision condition in $B_{\mathtt{OT},\text{corr}}$ into its negation, i.e., $m' = m_i$, we can similarly define the "attack-fail" language $A_{\mathtt{OT},\text{corr}}$; note that $B_{\mathtt{OT},\text{corr}}$ and $A^{\preccurlyeq}_{\mathtt{OT},\text{corr}}$ are disjoint. We observe that $B^{\text{ext}}_{\mathtt{OT},\text{corr}}$ is also decidable in polynomial time.

Definition 5.19 (Canonical Functionality $\mathcal{F}_{\text{corr}}$)**.** *The functionality $\mathcal{F}_{\text{corr}} \in \mathscr{F}_{\mathtt{OT}}$ equals $\mathcal{F}^{\mathsf{suppress},\mathsf{validate}}_{\mathtt{OT}}$, where (1)* suppress() *satisfies that for all x,* suppress$((\text{TRANSFER}, \mathbf{P}, x)) = x$*, (i.e., the same as in $\mathcal{F}^{\text{dum}}_{\mathtt{OT}}$), and (2)* validate$(w) = 0$ *if and only if $w \in B^{\text{ext}}_{\mathtt{OT},\text{corr}}$.*

5.2.2 Sender and receiver privacy

In order to capture sender privacy, we modify suppress() to withhold the sender's input from the adversary. This results in the following canonical functionality:

Definition 5.20 (Canonical Functionality $\mathcal{F}_{\text{ssec}}$)**.** *The functionality* $\mathcal{F}_{\text{ssec}} \in \mathscr{F}_{\text{OT}}$ *equals* $\mathcal{F}_{\text{OT}}^{\mathsf{suppress},\mathsf{validate}}$*, where (1)* $\mathsf{validate}() = 1$ *always, and (2)* $\mathsf{suppress}(\mathtt{a}) = (-)^{|m_0|+|m_1|}$*, for* $\mathtt{a} = (\text{TRANSFER}, \langle\langle S, R, sid\rangle, S\rangle, \langle m_0, m_1\rangle)$.

Similarly, we capture receiver privacy by suppressing the receiver's input:

Definition 5.21 (Canonical Functionality $\mathcal{F}_{\text{rsec}}$)**.** *The functionality* $\mathcal{F}_{\text{rsec}} \in \mathscr{F}_{\text{OT}}$ *equals* $\mathcal{F}_{\text{OT}}^{\mathsf{suppress},\mathsf{validate}}$*, where (1)* $\mathsf{validate}() = 1$ *always, and (2)* $\mathsf{suppress}(\mathtt{a}) = (-)^{|i|}$*, for* $\mathtt{a} = (\text{TRANSFER}, \langle\langle S, R, sid\rangle, R\rangle, i)$.

5.2.3 The canonical ideal oblivious transfer functionality

Based on the above, we obtain the canonical functionality $\mathcal{F}_{\text{OT}} = \mathcal{F}_{\text{ssec}} \wedge \mathcal{F}_{\text{rsec}} \wedge \mathcal{F}_{\text{corr}}$; see Figure 5.

Canonical Oblivious Transfer Functionality $\mathcal{F}_{\text{OT}}$

Action: TRANSFER

Well-formedness (WF_{OT}): Any TRANSFERRETURN symbol should be preceded by a TRANSFER symbol.

Public and Secret Outputs (PO_{OT}, SO_{OT}): For all w, $\mathtt{a}$, $\mathsf{PO}_{\text{OT}}(w, \mathtt{a}) = \epsilon$.
For all w, $\mathsf{SO}_{\text{OT}}(w, (\text{INFLTRANSFER}, \langle\langle S, R, sid\rangle, R\rangle)) = m_i$ if w contains $(\text{TRANSFER}, \langle\langle S, R, sid\rangle, S\rangle, \langle m_0, m_1\rangle)$ and $(\text{TRANSFER}, \langle\langle S, R, sid\rangle, R\rangle, i)$ and ϵ otherwise.

Suppress and Validate: (1) $\mathsf{suppress}()$ satisfies that for all x $\mathsf{suppress}((\text{TRANSFER}, \mathbf{P}, x)) = \epsilon$, and (2) $\mathsf{validate}(w) = 1$ if $w \notin B^{\text{ext}}_{\text{OT,corr}}$.

Figure 5: Ideal functionality for oblivious transfer based on the canonical functionality template.

5.2.4 Comparison to previous OT functionalities

Here we show that $\mathcal{F}_{\text{OT}}$ from Figure 5 is UC-equivalent to the oblivious transfer functionality as defined in [CLOS02], but different from the corresponding functionality given in [Can05]. The latter highlights a larger issue in the way ideal functionalities interact with the adversary in the UC framework. We first show the equivalence, followed by the treatment of the larger issue.

Call the ideal OT functionality from [CLOS02] $\mathcal{F}_{\text{OT}}^{\text{CLOS}}$ (refer to page 23, Figure 1 in [CLOS02]; we state the result more formally:

Proposition 5.22. *Functionality* $\mathcal{F}_{\text{OT}} = \mathcal{F}_{\text{corr}} \wedge \mathcal{F}_{\text{ssec}} \wedge \mathcal{F}_{\text{rsec}}$ *is UC-equivalent to* $\mathcal{F}_{\text{OT}}^{\text{CLOS}}$.

We now elaborate on the differences of our OT functionality with the one given in [Can05]. In [Can05], the notion of "delayed output" was introduced as a mechanism to enable the ideal-world adversary to delay the output of a certain action any amount time necessary to make the view of the environment indistinguishable to the real world's. This is important, as failing to provide such capability to the adversary may enable an impossibility result due to the existence of environments that can tell the real world from the ideal by simply observing the failure of the simulator to "synchronize" with the protocol flow in the real world.

Now, while the delayed output artifact successfully serves functionalities such as zero-knowledge and commitments (that turn out to be identical to our corresponding canonical versions), that is not the case for oblivious transfer. This is due to the fact that the basic action in oblivious transfer requires the input contribution from *both* the sender and the receiver prior to producing output. This asks for a more finely grained interaction between the ideal functionality and the ideal-world adversary. In our setting this is captured by the LEAKTRANSFER symbols that are sent to the adversary whenever a TRANSFER symbol is submitted by either the sender or the receiver.

In contrast, in the OT functionality of [Can05] such notifications are handled with two delayed outputs, something that forces the ideal functionality to wait when the receiver's input is submitted in case the sender has not submitted his input yet (cf. [Can05], Figure 25, page 108). Effectively, this creates a problem for OT protocols *where the receiver is supposed to send the first message*: in such case the environment can distinguish the real

world from the ideal world by activating the receiver without activating the sender and observing the network communication. This would not affect our canonical formulation of the OT functionality that notifies the ideal world adversary using the LEAKTRANSFER symbols whenever either party provides input.

Finally, some formulations of OT include an additional output produced by the functionality notifying the sender that the receiver has submitted its input [Wul07], which can be used to assist in the synchronization of the two parties. Such synchronization is not an essential aspect of oblivious transfer and as such it has not been included in previous formulations (see, e.g., [CLOS02]), nor it is here. We view this issue (relevant not only to OT but to any functionality that allows actions with inputs from more than one party) as a higher-layer consideration and not an aspect that is intended to be captured by an ideal formulation of the task.

5.3 Commitments

Following Figure 2, any canonical functionality for commitment, $\mathcal{F}_{\mathsf{COM}}$, is defined for two types of roles, the committer C and the verifier V, with two actions, COMMIT and OPEN. The $\mathsf{WF}_{\mathsf{COM}}$ predicate and $\mathsf{PO}_{\mathsf{COM}}, \mathsf{SO}_{\mathsf{COM}}$ mappings for $\mathcal{F}_{\mathsf{COM}}$ are defined in Figure 6. Based on these functions the dummy functionality $\mathcal{F}_{\mathsf{COM}}^{\mathrm{dum}}$ is defined (cf. Definition 3.3).

5.3.1 Correctness

In order to obtain the bad language for correctness, we observe that any committed value that is opened to should be accepted. Based on this, we identify the set of strings that are inconsistent with the correctness property as follows:

$$B_{\mathsf{COM},\mathrm{corr}} = \left\{ w \;\middle|\; \begin{array}{l} w = (\textsc{Commit}, \langle C, V, sid\rangle, m) \\ \quad (\textsc{CommitReturn}, \langle C, V, sid\rangle) \\ \quad (\textsc{Open}, \langle C, V, sid\rangle) \\ \quad (\textsc{OpenReturn}, \langle C, V, sid\rangle, \langle m, \phi\rangle) \\ \text{such that } \phi = 0 \end{array} \right\}$$

Following the methodology in Section 4.2, by changing the judge decision condition $\phi = 0$ in $B_{\mathsf{COM},\mathrm{corr}}$ into its negation, i.e., $\phi = 1$, we can similarly define the attack-fail language $A_{\mathsf{COM},\mathrm{corr}}$. Note that $B_{\mathsf{COM},\mathrm{corr}}$ and $A_{\mathsf{COM},\mathrm{corr}}^{\preccurlyeq}$ are disjoint. We observe that $B_{\mathsf{COM},\mathrm{corr}}^{\mathrm{ext}}$ is also decidable in polynomial time.

The class of ideal functionalities that corresponds to the correctness property can now be defined as follows:

Definition 5.23 (Canonical Functionality $\mathcal{F}_{\mathrm{corr}}$)**.** *The functionality $\mathcal{F}_{\mathrm{corr}} \in \mathscr{F}_{\mathsf{COM}}$ equals $\mathcal{F}_{\mathsf{COM}}^{\mathsf{suppress},\mathsf{validate}}$, where (1)* $\mathsf{suppress}()$ *is the same as in $\mathcal{F}_{\mathsf{COM}}^{\mathrm{dum}}$, and (2)* $\mathsf{validate}(w) = 0$ *if and only if $w \in B_{\mathsf{COM},\mathrm{corr}}^{\mathrm{ext}}$.*

5.3.2 Binding

The binding property basically states that any committed value that is opened to a different one should not be accepted. Based on this, we identify the set of strings that are inconsistent with the binding property as follows:

$$B_{\mathsf{COM},\mathrm{bind}} = \left\{ w \;\middle|\; \begin{array}{l} w = (\textsc{Commit}, \langle C, V, sid\rangle, m) \\ \quad (\textsc{CommitReturn}, \langle C, V, sid\rangle) \\ \quad (\textsc{Open}, \langle C, V, sid\rangle) \\ \quad (\textsc{OpenReturn}, \langle C, V, sid\rangle, \langle m', \phi\rangle) \\ \text{such that } m \neq m' \wedge \phi = 1 \end{array} \right\}$$

Following the methodology in Section 4.2, by changing the judge decision condition $m \neq m' \wedge \phi = 1$ in $B_{\mathsf{COM},\mathrm{bind}}$ into its negation, i.e., $m = m' \vee \phi = 0$, we can similarly define the attack-fail language $A_{\mathsf{COM},\mathrm{bind}}$. Note that $B_{\mathsf{COM},\mathrm{bind}}$ and $A_{\mathsf{COM},\mathrm{bind}}^{\preccurlyeq}$ are disjoint. We observe that $B_{\mathsf{COM},\mathrm{bind}}^{\mathrm{ext}}$ is also decidable in polynomial time.

We now define the class of ideal functionalities that corresponds to the binding property.

Definition 5.24 (Canonical Functionality $\mathcal{F}_{\mathrm{bind}}$)**.** *The functionality $\mathcal{F}_{\mathrm{bind}} \in \mathscr{F}_{\mathsf{COM}}$ equals $\mathcal{F}_{\mathsf{COM}}^{\mathsf{suppress},\mathsf{validate}}$ where (1)* $\mathsf{suppress}()$ *is the same as in $\mathcal{F}_{\mathsf{COM}}^{\mathrm{dum}}$, and (2)* $\mathsf{validate}(w) = 0$ *if and only if $w \in B_{\mathsf{COM},\mathrm{bind}}^{\mathrm{ext}}$.*

5.3.3 Hiding

For this property there is a natural hiding game. We apply our translation methodology to the game to obtain the corresponding ideal functionality class.

Definition 5.25 (Hiding). *A commitment scheme* $\Sigma(\mathtt{COM}) = \langle \mathtt{commit}, \mathtt{verify} \rangle$ *is* hiding *if for all PPT attackers* $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2)$*, it holds that*

$$\Pr\left[\begin{array}{l}(m_0, m_1, st) \leftarrow \mathsf{A}_1(1^\lambda); b \xleftarrow{r} \{0,1\}; \\ (c, \xi) \leftarrow \mathtt{commit}(m_b); b^* \leftarrow \mathsf{A}_2(st, c) : b^* = b \wedge m_0 \neq m_1\end{array}\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

The above definition can be modeled as a hiding game G_{hide} for the task $\mathtt{COM}$ as follows. The challenger C is allowed to use algorithms $\mathtt{commit}(), \mathtt{verify}()$ as oracles, and interacts with the attacker $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2)$. First A_1 produces a tuple $\langle m_0, m_1 \rangle$, where $m_0 \neq m_1$. In response, the challenger randomly chooses a bit b and queries the $\mathtt{commit}()$ oracle with m_b to obtain $\langle c, \xi \rangle$. Then, C sends c to A_2 to obtain b^* as a guess of b. The judge J decides that A wins the game if $b^* = b$. We next proceed to apply the methodology in Section 4.2.2.

Step 1. We construct an environment $\mathcal{Z}^{\mathsf{A}}_{\text{hide}}$ and the corresponding ideal world adversary $\mathcal{S}^{\Sigma}_{\text{hide}}$ based on the game G_{hide} described above. In order to simulate the game, the environment first picks C, V from the namespace at random as well as a random sid. Then it requests the corruption of the party V and simulates A_1 on input 1^λ. Once A_1 produces $\langle m_0, m_1 \rangle$, $\mathcal{Z}^{\mathsf{A}}_{\text{hide}}$ flips a random coin b, gives to C the symbol $(\textsc{Commit}, \langle C, V, sid \rangle, m_b)$ and waits for the transmission from C to V that contains the commitment c. Then, $\mathcal{Z}^{\mathsf{A}}_{\text{hide}}$ simulates A_2 on input c to obtain b^* and terminates with 1 if and only if $b = b^*$ and $m_0 \neq m_1$. The ideal world adversary $\mathcal{S}^{\Sigma}_{\text{hide}}$, whenever it receives $(\textsc{LeakCommit}, \langle C, V, sid \rangle, m)$, executes $\mathtt{commit}()$ on m and communicates the output of the protocol to the environment (similarly, it simulates the real world in any other respect).

Step 2. Based on the environment $\mathcal{Z}^{\mathsf{A}}_{\text{hide}}$ we define the functionality class that corresponds to the hiding game:

Definition 5.26 (Canonical Functionality $\mathcal{F}_{\text{hide}}$). *The functionality* $\mathcal{F}_{\text{hide}} \in \mathscr{F}_{\mathtt{COM}}$ *equals* $\mathcal{F}^{\mathsf{suppress},\mathsf{validate}}_{\mathtt{COM}}$*, where (1)* $\mathsf{validate}() = 1$ *always, and (2)* $\mathsf{suppress}(\mathsf{a}) = (-)^{|m|}$ *for* $\mathsf{a} = (\textsc{Commit}, \langle C, V, sid \rangle, m)$.

Based on Theorem 4.3, we have the following corollary:

Corollary 5.27. *If* $\pi_{\Sigma(\mathtt{COM})}$ *realizes some* $\mathcal{F} \gtrsim \mathcal{F}_{\text{hide}}$*, then* $\Sigma(\mathtt{COM})$ *satisfies hiding.*

We note that in this case the converse of the above corollary does not hold as hiding is not sufficiently strong to imply the UC-realization of $\mathcal{F}_{\text{hide}}$.

5.3.4 The canonical ideal commitment functionality

The (canonical) ideal commitment functionality $\mathcal{F}_{\mathtt{COM}} = \mathcal{F}_{\text{corr}} \wedge \mathcal{F}_{\text{bind}} \wedge \mathcal{F}_{\text{hide}}$, is instantiated in Figure 6, based on the canonical functionality template.

Canonical Commitment Functionality $\mathcal{F}_{\mathtt{COM}}$

Actions: $\textsc{Commit}$ and $\textsc{Open}$

Well-formedness ($\mathsf{WF}_{\mathtt{COM}}$): Symbols $\textsc{CommitReturn}$ should be preceded by $\textsc{Commit}$, $\textsc{OpenReturn}$ preceded by $\textsc{Open}$, $\textsc{Open}$ by $\textsc{Commit}$, and $\textsc{OpenReturn}$ by $\textsc{CommitReturn}$.

Public and Secret Outputs ($\mathsf{PO}_{\mathtt{COM}}$, $\mathsf{SO}_{\mathtt{COM}}$): For all w, a, $\mathsf{SO}_{\mathtt{COM}}(w, \mathsf{a}) = \epsilon$.
For all w, we have two cases (1) $\mathsf{PO}_{\mathtt{COM}}(w, (\textsc{Commit}, \langle C, V, sid \rangle, m)) = \epsilon$, and (2) $\mathsf{PO}_{\mathtt{COM}}(w, (\textsc{Open}, \langle C, V, sid \rangle)) = m$ if w contains $(\textsc{Commit}, \langle C, V, sid \rangle, m)$.

Suppress and Validate: (1) $\mathsf{suppress}()$ satisfies that for all m $\mathsf{suppress}((\textsc{Commit}, \mathbf{P}, m)) = \epsilon$, and $\mathsf{suppress}((\textsc{Open}, \mathbf{P})) = \epsilon$, and (2) $\mathsf{validate}(w) = 1$ if $w \notin B^{\text{ext}}_{\mathtt{COM},\text{corr}}$ and $w \notin B^{\text{ext}}_{\mathtt{COM},\text{bind}}$.

Figure 6: Ideal functionality for commitment based on the canonical functionality template.

Remark 5.28. $\mathcal{F}_{\mathtt{COM}}$ can be shown to be equivalent (in the sense of UC-emulation) to the commitment functionality as it appears in [Can05], in a way similar to the one used to show the equivalence between our canonical OT functionality and the one in [CLOS02]. As such, $\mathcal{F}_{\mathtt{COM}}$ is unrealizable in the plain model. Interestingly, the pairwise conjunction of its constituent functionalities is in fact realizable (refer to Figure 3 for the realizability "horizon"). The three simple protocols are given below; it is easy to see they realize the corresponding conjunctions.

- $\mathcal{F}_{\text{corr}} \wedge \mathcal{F}_{\text{bind}}$: Consider the following protocol:
  - ⋆ Upon receiving $(\text{COMMIT}, \mathbf{P}, m)$ from the environment $\mathcal{Z}$, party C sends m as the commitment value to party V (through the real world adversary $\mathcal{A}$), and whenever party V receives commitment value m from party C (through $\mathcal{A}$), it returns $(\text{COMMITRETURN}, \mathbf{P})$ to $\mathcal{Z}$;
  - ⋆ upon receiving $(\text{OPEN}, \mathbf{P})$ from $\mathcal{Z}$, party C sends m as the opening to party V (through $\mathcal{A}$), and whenever party V receives opening value m from party C (through $\mathcal{A}$), it returns $(\text{OPENRETURN}, \mathbf{P}, \langle m, 1\rangle)$ to $\mathcal{Z}$.
- $\mathcal{F}_{\text{corr}} \wedge \mathcal{F}_{\text{hide}}$: Consider the following protocol:
  - ⋆ Upon receiving $(\text{COMMIT}, \mathbf{P}, m)$ from $\mathcal{Z}$, party C sends a committing notice as the commitment value to party V (through $\mathcal{A}$), and whenever party V receives such notice from party C (through $\mathcal{A}$), it returns $(\text{COMMITRETURN}, \mathbf{P})$ to $\mathcal{Z}$;
  - ⋆ upon receiving $(\text{OPEN}, \mathbf{P})$ from $\mathcal{Z}$, party C sends m as the opening to party V (through $\mathcal{A}$), and whenever party V receives opening value m from party C (through $\mathcal{A}$), it returns $(\text{OPENRETURN}, \mathbf{P}, \langle m, 1\rangle)$ to $\mathcal{Z}$.
- $\mathcal{F}_{\text{bind}} \wedge \mathcal{F}_{\text{hide}}$: The protocol is very similar to the protocol above except for the following difference: here party V rejects the opening, while in the protocol above, it accepts the opening:
  - ⋆ Upon receiving $(\text{COMMIT}, \mathbf{P}, m)$ from $\mathcal{Z}$, party C sends a committing notice as the commitment value to party V (through $\mathcal{A}$), and whenever party V receives such notice from party C (through $\mathcal{A}$), it returns $(\text{COMMITRETURN}, \mathbf{P})$ to $\mathcal{Z}$;
  - ⋆ upon receiving $(\text{OPEN}, \mathbf{P})$ from $\mathcal{Z}$, party C sends m as the opening to party V (through $\mathcal{A}$), and whenever party V receives opening value m from party C (through $\mathcal{A}$), it returns $(\text{OPENRETURN}, \mathbf{P}, \langle m, 0\rangle)$ to $\mathcal{Z}$.

5.4 Zero-knowledge proofs

Following Figure 2, the canonical functionality for zero-knowledge [GMR89, BG92], $\mathcal{F}_{\mathtt{ZK}}^{R}$, is defined for two types of roles, the prover P and the verifier V, with a single action PROVE. We denote the zero-knowledge proof functionality class as $\mathscr{F}_{\mathtt{ZK}}^{R}$. (Sometimes we omit the reference to R in the notation for simplicity.) The $\mathsf{WF}_{\mathtt{ZK}}$ predicate for $\mathcal{F}_{\mathtt{ZK}}^{R}$, requires that a PROVE symbol should precede PROVERETURN. The public output $\mathsf{PO}_{\mathtt{ZK}}$ returns $\langle x, \phi\rangle$ whenever $(\text{PROVE}, \langle P, V, sid\rangle, \langle x, w\rangle)$ is in the history, where $\phi = 1$ if and only if $\langle x, m\rangle$ belongs to the relation that parameterizes the zero-knowledge task, and $\phi = 0$ otherwise. Based on the above the dummy functionality $\mathcal{F}_{\mathtt{ZK}}^{\text{dum}}$ is defined (cf. Definition 3.3).

5.4.1 Completeness

In order to obtain the bad language for completeness, we observe that any $(x, m) \in R$ should be accepted; the set of strings that are inconsistent with the completeness property are as follows:

$$B_{\mathtt{ZK},\text{comp}} = \left\{ w \;\middle|\; \begin{array}{l} w = (\text{PROVE}, \langle P, V, sid\rangle, \langle x, m\rangle) \\ \qquad (\text{PROVERETURN}, \langle P, V, sid\rangle, \langle x, \phi\rangle) \\ \text{such that } (x, m) \in R \wedge \phi = 0 \end{array} \right\}$$

Following the methodology in Section 4.2, by changing the judge decision condition $(x, m) \in R \wedge \phi = 0$ in $B_{\mathtt{ZK},\text{comp}}$ into its negation, i.e., $(x, m) \notin R \vee \phi = 1$, we can similarly define the attack-fail language $A_{\mathtt{ZK},\text{comp}}$. Note that $B_{\mathtt{ZK},\text{comp}}$ and $A_{\mathtt{ZK},\text{comp}}^{\preccurlyeq}$ are disjoint. We observe that $B_{\mathtt{ZK},\text{comp}}^{\text{ext}}$ is also decidable in polynomial time.

The class of ideal functionalities that corresponds to the completeness property is as follows:

Definition 5.29 (Canonical Functionality $\mathcal{F}^R_{\text{comp}}$)**.** *The functionality* $\mathcal{F}^R_{\text{comp}} \in \mathscr{F}^R_{\text{ZK}}$ *equals* $\mathcal{F}^{\mathsf{suppress},\mathsf{validate}}_{\text{ZK}}$ *where (1)* $\mathsf{suppress}()$ *is same as in* $\mathcal{F}^{\text{dum}}_{\text{ZK}}$*, and (2)* $\mathsf{validate}(w) = 0$ *if and only if* $w \in B^{\text{ext}}_{\text{ZK,comp}}$.

5.4.2 Soundness

In order to obtain the bad language for soundness, we observe that any $(x, m) \notin R$ should not be accepted; therefore, the set of strings that are inconsistent with the completeness property are:

$$B_{\text{ZK,sound}} = \left\{ w \;\middle|\; \begin{array}{l} w = (\textsc{Prove}, \langle P, V, sid\rangle, \langle x, m\rangle) \\ \quad\quad (\textsc{ProveReturn}, \langle P, V, sid\rangle, \langle x, \phi\rangle) \\ \text{such that } (x, m) \notin R \wedge \phi = 1 \end{array} \right\}$$

Following the methodology in Section 4.2, by changing the judge decision condition $(x, m) \notin R \wedge \phi = 1$ in $B_{\text{ZK,sound}}$ into its negation, i.e., $(x, m) \in R \vee \phi = 0$, we can similarly define the attack-fail language $A_{\text{ZK,sound}}$. Note that $B_{\text{ZK,sound}}$ and $A^{\preccurlyeq}_{\text{ZK,sound}}$ are disjoint. We observe that $B^{\text{ext}}_{\text{ZK,sound}}$ is also decidable in polynomial time.

We next define the class of ideal functionalities for soundness:

Definition 5.30 (Canonical Functionality $\mathcal{F}^R_{\text{sound}}$)**.** *The functionality* $\mathcal{F}^R_{\text{sound}} \in \mathscr{F}^R_{\text{ZK}}$ *equals* $\mathcal{F}^{\mathsf{suppress},\mathsf{validate}}_{\text{ZK}}$ *where (1)* $\mathsf{suppress}()$ *is same as in* $\mathcal{F}^{\text{dum}}_{\text{ZK}}$*, and (2)* $\mathsf{validate}(w) = 0$ *if and only if* $w \in B^{\text{ext}}_{\text{ZK,sound}}$.

We note that the soundness notion that $\mathcal{F}^R_{\text{sound}}$ captures is the "strong" one, as stipulated by the knowledge extraction property.

5.4.3 Zero-knowledge

To capture the zero-knowledge property, we suppress the input from the prover; based on the template in Figure 2, we obtain the following functionality.

Definition 5.31 (Canonical Functionality $\mathcal{F}^R_{\text{zk}}$)**.** *The functionality* $\mathcal{F}^R_{\text{zk}} \in \mathscr{F}^R_{\text{ZK}}$ *equals* $\mathcal{F}^{\mathsf{suppress},\mathsf{validate}}_{\text{ZK}}$*, where (1)* $\mathsf{validate}() = 1$ *always, and (2)* $\mathsf{suppress}(\mathsf{a}) = (-)^{|x|+|m|}$ *for* $\mathsf{a} = (\textsc{Prove}, \langle P, V, sid\rangle, \langle x, m\rangle)$.

5.4.4 The canonical ideal ZK functionality

The ZK functionality equals $\mathcal{F}^R_{\text{comp}} \wedge \mathcal{F}^R_{\text{sound}} \wedge \mathcal{F}^R_{\text{zk}}$, which turns out to be equivalent (in the sense of UC-emulation) to the zero-knowledge functionality as it appears in [Can05].

Canonical Zero-Knowledge Functionality $\mathcal{F}^R_{\text{ZK}}$

Action: PROVE

Well-formedness (WF_{ZK}): Symbols PROVERETURN should be preceded by PROVE.

Public and Secret Outputs (PO_{ZK}, SO_{ZK}): For all w, a, $\mathsf{SO}_{\text{ZK}}(w, \mathsf{a}) = \epsilon$.
For all w, $\mathsf{PO}_{\text{ZK}}(w, (\textsc{Prove}, \langle P, V, sid\rangle, \langle x, m\rangle)) = \langle x, \phi\rangle$, where $\phi = 1$ iff $(x, m) \in R$ and $\phi = 0$ otherwise.

Suppress and Validate: (1) $\mathsf{suppress}()$ satisfies that for all statement-witness pair (x, m), $\mathsf{suppress}((\textsc{Prove}, \mathbf{P}, \langle x, m\rangle)) = \epsilon$, and (2) $\mathsf{validate}(w) = 1$ if $w \notin B^{\text{ext}}_{\text{ZK,comp}}$ and $w \notin B^{\text{ext}}_{\text{ZK,sound}}$.

Figure 7: Ideal functionality for zero-knowledge, based on the canonical functionality template.

Acknowledgements. The authors thank Manoj Prabhakaran for helpful comments.

References

[Bea91] Donald Beaver. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. *J. Cryptology*, 4(2):75–122, 1991.

[BG92] Mihir Bellare and Oded Goldreich. On defining proofs of knowledge. In Ernest F. Brickell, editor, *CRYPTO*, volume 740 of *LNCS*, pages 390–420. Springer, 1992.

[BH04] Michael Backes and Dennis Hofheinz. How to break and repair a universally composable signature functionality. In Kan Zhang and Yuliang Zheng, editors, *ISC*, volume 3225 of *LNCS*, pages 61–72. Springer, 2004.

[BPW04] Michael Backes, Birgit Pfitzmann, and Michael Waidner. Low-level ideal signatures and general integrity idealization. In *ISC*, pages 39–51, 2004.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[Can04] Ran Canetti. Universally composable signature, certification, and authentication. In *CSFW*, pages 219–235, 2004. Full version at http://eprint.iacr.org/2003/239/.

[Can05] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Cryptology ePrint Archive, Report 2000/067*, December 2005. Latest version at http://eprint.iacr.org/2000/067/.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC*, volume 4392 of *LNCS*, pages 61–85. Springer, 2007.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *STOC*, pages 494–503. ACM, 2002. Full version at http://eprint.iacr.org/2002/140/.

[Cré87] Claude Crépeau. Equivalence between two flavours of oblivious transfers. In Carl Pomerance, editor, *CRYPTO*, volume 293 of *LNCS*, pages 350–354. Springer, 1987.

[DDM+06] Anupam Datta, Ante Derek, John C. Mitchell, Ajith Ramanathan, and Andre Scedrov. Games and the impossibility of realizable ideal functionality. In Shai Halevi and Tal Rabin, editors, *TCC*, volume 3876 of *LNCS*, pages 360–379. Springer, 2006.

[DDN00] Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM J. Comput.*, 30(2):391–437, 2000.

[EGL85] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.

[GKZ09] Juan Garay, Aggelos Kiayias, and Hong-Sheng Zhou. Sound and fine-grain specification of cryptographic tasks. Cryptology ePrint Archive, Report 2008/132, February 15, 2009.

[GL90] Shafi Goldwasser and Leonid A. Levin. Fair computation of general functions in presence of immoral majority. In Alfred Menezes and Scott A. Vanstone, editors, *CRYPTO*, volume 537 of *LNCS*, pages 77–93. Springer, 1990.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, pages 218–229. ACM, 1987.

[Küs06] Ralf Küsters. Simulation-based security with inexhaustible interactive turing machines. In *CSFW*, pages 309–320, 2006.

[LPV09] Huijia Lin, Rafael Pass, and Muthuramakrishnan Venkitasubramaniam. A unified framework for concurrent security: universal composability from stand-alone non-malleability. In *STOC*, pages 179–188. ACM, 2009.

[MR91] Silvio Micali and Phillip Rogaway. Secure computation (abstract). In Joan Feigenbaum, editor, *CRYPTO*, volume 576 of *LNCS*, pages 392–404. Springer, 1991.

[Pat05] Akshay Patil. On symbolic analysis of cryptographic protocols. In *Master's thesis*. Massachusetts Institute of Technology, 2005.

[PS04] Manoj Prabhakaran and Amit Sahai. New notions of security: achieving universal composability without trusted setup. In *STOC*, pages 242–251. ACM, 2004.

[PW01] Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *IEEE Symposium on Security and Privacy*, 2001.

[Rab81] Michael Rabin. How to exchange secrets by oblivious transfer. In *Technical Report TR-81*. Harvard University, 1981.

[Wul07] Jürg Wullschleger. Oblivious-transfer amplification. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *LNCS*, pages 555–572. Springer, 2007. Full version at http://arxiv.org/abs/cs.CR/0608076.

A Digital Signatures (cont'd)

In this section we give an alternative description of the bad language for the unforgeability game, and then we present the attack-fail language. Recall that, for simplicity, in Section 5.1 we only present a *subset* of the attack-fail language. Based on the "full" bad language and the attack-fail language presented here, we derive an extended bad language which is then shown to be equivalent to the extended bad language presented in Section 5.1.

Similar to the transformation in the Section 5.1, the GMR unforgeability definition can be formulated as a consistency game G_{uf} for the task SIG as follows: the challenger C uses algorithms $\texttt{gen}(), \texttt{sign}(), \texttt{verify}()$ as oracles, and interacts with the adversary A: the challenger C queries the $\texttt{gen}()$ oracle and obtains $\langle sk, vk\rangle$, and then sends such vk to A; each time upon receiving m_i from the A, the challenger C queries the $\texttt{sign}()$ oracle with m_i and obtains σ_i, and then returns σ to A; upon receiving from A a pair $\langle m', \sigma'\rangle$, C queries the $\texttt{verify}()$ oracle with $\langle m', \sigma', vk'\rangle$ and obtains the verification result. The judge J decides that A wins the game if m' has never been queried before, $vk' = vk$, and the verification result is 1.

Step 1. Based on the game G_{uf} described above, we can construct an environment $\mathcal{Z}_{\text{uf}}^{\mathsf{A}}$ and the corresponding ideal world adversary $\mathcal{S}_{\text{uf}}^{\Sigma}$ as follows. In order to simulate the game, the environment first picks S and V from the

namespace at random as well as a random sid. The environment sends $(\text{KEYGEN}, \langle S, sid\rangle)$ to party S and receives $(\text{KEYGENRETURN}, \langle S, sid\rangle, vk)$; then the environment simulates A on input vk; when A queries m_i to its signing oracle, the environment sends $(\text{SIGN}, \langle S, sid\rangle, m_i)$ to party S and returns the output of S to A. Once A outputs a pair $\langle m', \sigma'\rangle$, the environment sends $(\text{VERIFYRETURN}, \langle V, sid\rangle, \langle m', \sigma', vk'\rangle)$ to some party V and receives the verification result ϕ. In the case that m' has never been queried and $\phi = 1$, and $vk' = vk$, the environment terminates with 1; otherwise with 0.

We next define the ideal-world adversary $\mathcal{S}^{\Sigma}_{\text{uf}}$. Each time $\mathcal{S}^{\Sigma}_{\text{uf}}$ receives $(\text{LEAKKEYGEN}, \langle S, sid\rangle)$ from the ideal functionality, it runs $(vk, sk) \leftarrow \mathtt{gen}(1^{\lambda})$ and sends $(\text{INFLKEYGEN}, \langle S, sid\rangle, vk)$ to the functionality. When $\mathcal{S}^{\Sigma}_{\text{uf}}$ receives $(\text{LEAKSIGN}, \langle S, sid\rangle, m_i)$ from the ideal functionality, it runs $\sigma \leftarrow \mathtt{sign}(vk, sk, m_i)$, and sends $(\text{INFLSIGN}, \langle S, sid\rangle, \sigma_i)$ to the functionality. When $\mathcal{S}^{\Sigma}_{\text{uf}}$ receives $(\text{LEAKVERIFY}, \langle V, sid\rangle, \langle m', \sigma', vk'\rangle)$ from the ideal functionality, it runs $\phi \leftarrow \mathtt{verify}(vk', sk, m', \sigma')$, and sends $(\text{INFLVERIFY}, \langle V, sid\rangle, \phi)$ to the functionality.

Step 2. For any adversary A and signature scheme Σ we define $L_{\mathcal{F}^{\text{dum}}_{\text{SIG}}, \mathcal{Z}^{\mathsf{A}}_{\text{uf}}, \mathcal{S}^{\Sigma}_{\text{uf}}}$ (cf. Section 3.2) with $\mathcal{Z}^{\mathsf{A}}_{\text{uf}}, \mathcal{S}^{\Sigma}_{\text{uf}}$ as defined in step 1. We next define the set of strings $B_{\mathtt{SIG},\text{uf}}$ as the subset of $\bigcup_{\mathsf{A},\Sigma} L_{\mathcal{F}^{\text{dum}}_{\text{SIG}}, \mathcal{Z}^{\mathsf{A}}_{\text{uf}}, \mathcal{S}^{\Sigma}_{\text{uf}}}$ that contains exactly those strings for which the environment returns 1.

$$B'_{\mathtt{SIG},\text{uf}} = \left\{ w \;\middle|\; \begin{array}{l} w = (\text{KEYGEN}, \langle S, sid\rangle) \\ \quad (\text{KEYGENRETURN}, \langle S, sid\rangle, vk) \\ \quad (\text{SIGN}, \langle S, sid\rangle, m_1) \\ \quad (\text{SIGNRETURN}, \langle S, sid\rangle, \sigma_1) \\ \quad \ldots \\ \quad (\text{SIGN}, \langle S, sid\rangle, m_\ell) \\ \quad (\text{SIGNRETURN}, \langle S, sid\rangle, \sigma_\ell) \\ \quad (\text{VERIFY}, \langle V, sid\rangle, \langle m', \sigma', vk'\rangle) \\ \quad (\text{VERIFYRETURN}, \langle V, sid\rangle, \phi) \\ \text{such that} \\ \phi = 1 \wedge m' \notin \{m_1, \ldots, m_\ell\} \wedge vk' = vk \end{array} \right\}$$

Following the methodology in Section 4.2, by changing the judge decision condition into its negation, i.e., $\phi = 0 \vee m' \in \{m_1, \ldots, m_\ell\} \vee vk' \neq vk$, we can define the "attack-fail" language $A'_{\mathtt{SIG},\text{uf}}$. We note that it is easy to check that $B'_{\mathtt{SIG},\text{uf}} = B_{\mathtt{SIG},\text{uf}}$ while $A_{\mathtt{SIG},\text{uf}} \subset A'_{\mathtt{SIG},\text{uf}}$. We also observe that $B_{\mathtt{SIG},\text{uf}}$ is disjointed with $A'_{\mathtt{SIG},\text{uf}} \backslash A_{\mathtt{SIG},\text{uf}}$. Recall that the extended bad language according to the unforgeability property consists of the strings w of $L_{\mathcal{F}^{\text{dum}}_{\text{SIG}}}$ that contain as a subsequence a string w' satisfies the condition that $w' \in B'_{\mathtt{SIG},\text{uf}} \wedge w' \notin A'^{\preccurlyeq}_{\mathtt{SIG},\text{uf}} \cap \{w\}^{\preccurlyeq}$; this condition can further be simplified as $w' \in B_{\mathtt{SIG},\text{uf}} \wedge w' \notin A^{\preccurlyeq}_{\mathtt{SIG},\text{uf}} \cap \{w\}^{\preccurlyeq}$ because $B'_{\mathtt{SIG},\text{uf}} = B_{\mathtt{SIG},\text{uf}}$ and $B_{\mathtt{SIG},\text{uf}}$ is disjointed with $A'_{\mathtt{SIG},\text{uf}} \backslash A_{\mathtt{SIG},\text{uf}}$ as observed. So the extended bad language defined here is same as the one in the body part.

We can also obtain the extended bad languages from the attack-fail languages as demonstrated here for the other consistency properties. Similarly to the case of unforgeability, for simplicity we obtain the extended bad language for each consistency property from a *subset* of the attack-fail language, which can then be shown to be equivalent to the one from the full attack-fail language.

B Proofs

Proof of Proposition 3.4

Proof. To prove the proposition, we need to show that (1) $\mathscr{F}_T$ is associative, (2) $\mathscr{F}_T$ has an identity $\mathcal{F}^{\text{dum}}_T$, and (3) $\mathscr{F}_T$ is commutative.

First we show $\wedge$ is associative. That is for all $\mathcal{F}_1 = \mathcal{F}_T^{\mathsf{suppress}_1, \mathsf{validate}_1}$, $\mathcal{F}_2 = \mathcal{F}_T^{\mathsf{suppress}_2, \mathsf{validate}_2}$, and $\mathcal{F}_3 = \mathcal{F}_T^{\mathsf{suppress}_3, \mathsf{validate}_3} \in \mathscr{F}_T$, we need to show $(\mathcal{F}_1 \wedge \mathcal{F}_2) \wedge \mathcal{F}_3 = \mathcal{F}_1 \wedge (\mathcal{F}_2 \wedge \mathcal{F}_3)$. For any $\mathtt{a} = (\text{ACTION}, \mathbf{P}, x)$, we let $\mathsf{suppress}_{12}(\mathtt{a}) = \mathsf{suppress}_1(\text{ACTION}, \mathbf{P}, (\mathsf{suppress}_2(\mathtt{a}))$ and let $\mathsf{suppress}_{23}(\mathtt{a}) = \mathsf{suppress}_2(\text{ACTION}, \mathbf{P}, (\mathsf{suppress}_3(\mathtt{a}))$.

For $(\mathcal{F}_1 \wedge \mathcal{F}_2) \wedge \mathcal{F}_3$, we have $\mathsf{suppress}_{(12)3}(\mathtt{a}) = \mathsf{suppress}_{12}(\text{ACTION}, \mathbf{P}, (\mathsf{suppress}_3(\mathtt{a}))) = \mathsf{suppress}_1(\text{ACTION}, \mathbf{P}, (\mathsf{suppress}_2(\text{ACTION}, \mathbf{P}, (\mathsf{suppress}_3(\mathtt{a}))))$. For $\mathcal{F}_1 \wedge (\mathcal{F}_2 \wedge \mathcal{F}_3)$, we have $\mathsf{suppress}_{1(23)}(\mathtt{a}) = \mathsf{suppress}_1(\text{ACTION}, \mathbf{P}, (\mathsf{suppress}_{23}(\mathtt{a}))) = \mathsf{suppress}_1(\text{ACTION}, \mathbf{P}, (\mathsf{suppress}_2(\text{ACTION}, \mathbf{P}, (\mathsf{suppress}_3(\mathtt{a})))))$. Therefore $\mathsf{suppress}_{(12)3}(\mathtt{a}) = \mathsf{suppress}_{1(23)}(\mathtt{a})$. Further validate predicates follow logical conjunction operation, and we have $\mathsf{validate}_{(12)3}() = (\mathsf{validate}_1() \wedge \mathsf{validate}_2()) \wedge \mathsf{validate}_3() = \mathsf{validate}_1() \wedge (\mathsf{validate}_2() \wedge \mathsf{validate}_3()) = \mathsf{validate}_{1(23)}()$. Together, we have $(\mathcal{F}_1 \wedge \mathcal{F}_2) \wedge \mathcal{F}_3 = \mathcal{F}_1 \wedge (\mathcal{F}_2 \wedge \mathcal{F}_3)$.

Second, we show $\mathscr{F}_T$ has an identity $\mathcal{F}_T^{\text{dum}}$. Let $\mathcal{F}_0 = \mathcal{F}_T^{\text{dum}} = \mathcal{F}_T^{\mathsf{suppress}_0, \mathsf{validate}_0}$, then we have for any $\mathtt{a} = (\text{ACTION}, \mathbf{P}, x)$, $\mathsf{suppress}_0(\mathtt{a}) = x$, and $\mathsf{validate}_0() = 1$. We need to show for all $\mathcal{F}_1 = \mathcal{F}_T^{\mathsf{suppress}_1, \mathsf{validate}_1} \in \mathscr{F}_T$, it holds that $\mathcal{F}_1 \wedge \mathcal{F}_0 = \mathcal{F}_1 = \mathcal{F}_0 \wedge \mathcal{F}_1$. Now we have $\mathsf{suppress}_{10}(\mathtt{a}) = \mathsf{suppress}_1(\text{ACTION}, \mathbf{P}, \mathsf{suppress}_0(\mathtt{a})) = \mathsf{suppress}_1(\text{ACTION}, \mathbf{P}, x) = \mathsf{suppress}_1(\mathtt{a})$, and $\mathsf{suppress}_{01}(\mathtt{a}) = \mathsf{suppress}_0(\text{ACTION}, \mathbf{P}, \mathsf{suppress}_1(\mathtt{a})) = \mathsf{suppress}_1(\mathtt{a})$; that is $\mathsf{suppress}_{10}(\mathtt{a}) = \mathsf{suppress}_1(\mathtt{a}) = \mathsf{suppress}_{10}(\mathtt{a})$. Further we have $\mathsf{validate}_{10}() = \mathsf{validate}_1() \wedge \mathsf{validate}_0() = \mathsf{validate}_1() \wedge 1 = \mathsf{validate}_1()$, and $\mathsf{validate}_{01}() = \mathsf{validate}_0() \wedge \mathsf{validate}_1() = 1 \wedge \mathsf{validate}_1() = \mathsf{validate}_1()$; that is $\mathsf{validate}_{10}() = \mathsf{validate}_1() = \mathsf{validate}_{01}()$. Together, we have $\mathcal{F}_1 \wedge \mathcal{F}_0 = \mathcal{F}_1 = \mathcal{F}_0 \wedge \mathcal{F}_1$.

Third, we show $\mathscr{F}_T$ is commutative. That is for all $\mathcal{F}_1 = \mathcal{F}_T^{\mathsf{suppress}_1, \mathsf{validate}_1}$, $\mathcal{F}_2 = \mathcal{F}_T^{\mathsf{suppress}_2, \mathsf{validate}_2} \in \mathscr{F}_T$, we need to show $\mathcal{F}_1 \wedge \mathcal{F}_2 = \mathcal{F}_2 \wedge \mathcal{F}_1$. Note that for any $\mathtt{a} = (\text{ACTION}, \mathbf{P}, x)$, the $\mathsf{suppress}()$ functions substitute with "$-$" the same locations of x. We let $x_{12} = \mathsf{suppress}_{12}(\mathtt{a}) = \mathsf{suppress}_1(\text{ACTION}, \mathbf{P}, (\mathsf{suppress}_2(\mathtt{a}))$ and let $x_{21} = \mathsf{suppress}_{21}(\mathtt{a}) = \mathsf{suppress}_2(\text{ACTION}, \mathbf{P}, (\mathsf{suppress}_1(\mathtt{a}))$. The same locations in both x_{12} and x_{21} are substituted with "$-$" from x. So $x_{12} = x_{21}$, i.e., $\mathsf{suppress}_{12}(\mathtt{a}) = \mathsf{suppress}_{21}(\mathtt{a})$. Further we know $\mathsf{validate}_1() \wedge \mathsf{validate}_2() = \mathsf{validate}_2() \wedge \mathsf{validate}_1()$ because validate predicates follow logical conjunction operation. Together, we have $\mathcal{F}_1 \wedge \mathcal{F}_2 = \mathcal{F}_2 \wedge \mathcal{F}_1$.

Together we show $(\mathscr{F}_T, \wedge)$ is a commutative monoid with the dummy functionality $\mathcal{F}_T^{\text{dum}}$ as the identity element. □

Proof of Theorem 3.6

Proof. Consider a task T, and its well-formedness predicate WF_T. We construct a scheme Σ that implements T such that π_Σ realizes the dummy functionality $\mathcal{F}_T^{\text{dum}}$. We first give description for π_T, then we design the scheme Σ; the protocol π_Σ will be obtained by implementing all actions of π_T with the algorithms of Σ. A π_T entity P maintains an array history, initially empty, which is used to record the entity's action symbols. In particular, when P receives a symbol $(\text{ACTION}, \mathbf{P}, x)$ from the environment, it records the symbol into its history, runs the predicate WF_T over history, and if the predicate returns 0, then the input is ignored, and the input will be removed from its history. Whenever required by the action, the π_T entity returns an output symbol $(\text{ACTIONRETURN}, \mathbf{P}, y)$, using the WF_T predicate to ensure well-formedness. We next describe the scheme Σ implementing the cryptographic task T. Recall that for each action T specifies a domain and range; given that we are only interested in designing a protocol realizing the dummy functionality we will simply define each action of Σ to map every input of the action domain $D_i^{(\lambda)}$ to an element of the action range $R_i^{(\lambda)}$. This captures the case of a non-interactive action. For interactive actions, say between two parties, Σ provides a two party protocol where the two parties coordinate according to the input-output behavior of the action. This completes the description of Σ that together with π_T defines the protocol π_Σ.

Next, we construct an ideal world adversary $\mathcal{S}$ such that no environment $\mathcal{Z}$ can distinguish an execution involving π_Σ and the real world adversary from an execution of $\mathcal{F}_T^{\text{dum}}$ and the ideal world adversary. The construction of $\mathcal{S}$ is as follows: $\mathcal{S}$ will simply perform a faithful simulation of the real world execution with the protocol π_Σ and the real-world adversary. This is possible as the canonical dummy functionality relays all (valid) I/O from the environment without any modifications. We next prove that no environment $\mathcal{Z}$ can distinguish the ideal from the real world for the above simulator $\mathcal{S}$ and in fact the simulation is perfect.

Observe that the only difference between the real world execution and the ideal world execution is the fact that the verification of the well-formedness predicate in the real world is distributed amongst the parties whereas in the ideal world it is handled by the canonical functionality. Observe that if the combined history of all parties in an ideal world execution is well-formed then the local history of each party in the real world will also be well-

formed (as the same WF_T predicate is used globally and locally and the predicate is only sensitive in the order of symbols). Note that the reverse direction is not necessarily true; indeed a set of well-formed local histories may not be composed to a global history that is well-formed (and this may provide an opportunity for an adversarial environment to distinguish the real from the ideal world). Nevertheless, this is not the case due to the fact that a Σ scheme, specifically the coordination component of the protocol implementation of interactive actions, will ensure that the composition (according to the real order of events as induced by the adversary) of the local histories of all parties in a real world execution will result in a well-formed global history.

In the case of corrupted parties observe that the composed global history of a real world execution might cease to be well-formed as it may not include the local histories of corrupted parties (which are handled internally by the adversary). This discrepancy, however, will not result in any distinguishing advantage as the simulator $\mathcal{S}$ has the power to insert symbols in the canonical functionality's history that follow the actions of corrupted parties and thus maintain the well-formedness of the functionality's history.

Based on the above we conclude that the ideal world adversary $\mathcal{S}$ is performing a perfect simulation of the ideal world when interacting with $\mathcal{F}_T^{\text{dum}}$ and thus π_Σ is a UC-realization of $\mathcal{F}_T^{\text{dum}}$. □

Proof of Theorem 3.7

Proof. Let π be a protocol that UC-realizes $\mathcal{F}$ and let $\mathcal{F}'$ be any functionality such that $\mathcal{F}' \lesssim \mathcal{F}$ which means that $\mathcal{F} = \mathcal{F}' \wedge \mathcal{F}''$ for some $\mathcal{F}'' \in \mathscr{F}_T$. Let $\mathcal{F}' = \mathcal{F}_T^{\mathsf{suppress}_1,\mathsf{validate}_1}, \mathcal{F}'' = \mathcal{F}_T^{\mathsf{suppress}_2,\mathsf{validate}_2} \in \mathscr{F}_T$. To prove the theorem, it suffices to prove the following statement that any protocol π that UC-realizes $\mathcal{F}$ also UC-realizes $\mathcal{F}'$.

To prove that π UC-realizes $\mathcal{F}'$, we need to show that for any $\mathcal{A}'$ there is an ideal world adversary $\mathcal{S}'$ such that for all $\mathcal{Z}'$, $\text{IDEAL}_{\mathcal{F}',\mathcal{S}',\mathcal{Z}'} \approx \text{REAL}_{\pi,\mathcal{A}',\mathcal{Z}'}$. Notice that based on the condition that protocol π realizes $\mathcal{F}$, for any $\mathcal{A}$ there is an ideal world adversary $\mathcal{S}$ such that for all $\mathcal{Z}$, $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \approx \text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$.

Given a real world adversary $\mathcal{A}'$ for the protocol π, there exists an $\mathcal{S}$ from the premise of the theorem that simulates it in the ideal world interacting with $\mathcal{F}$. We construct an $\mathcal{S}'$ that interacts with $\mathcal{F}'$ as follows: $\mathcal{S}'$ simulates $\mathcal{S}$ in its interface with the functionality $\mathcal{F}'$ with the following modification: each time when $\mathcal{F}'$ has input $\mathtt{a} = (\text{ACTION}, \mathbf{P}, x)$ it gives to the adversary the symbol $(\text{LEAKACTION}, \mathbf{P}, x_1)$ where $x_1 = \mathsf{suppress}_1(\mathtt{a})$; given this symbol, $\mathcal{S}'$ computes $x_2 = \mathsf{suppress}_2(\text{ACTION}, \mathbf{P}, x_1)$ and gives the symbol $(\text{LEAKACTION}, \mathbf{P}, x_2)$ to $\mathcal{S}$. This completes the description of $\mathcal{S}'$.

Given an environment $\mathcal{Z}'$ we will show that $\text{IDEAL}_{\mathcal{F}',\mathcal{S}',\mathcal{Z}'} \approx \text{REAL}_{\pi,\mathcal{A}',\mathcal{Z}'}$. From the premise of the theorem we know that $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}'} \approx \text{REAL}_{\pi,\mathcal{A}',\mathcal{Z}'}$, thus it suffices to show $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}'} \approx \text{IDEAL}_{\mathcal{F}',\mathcal{S}',\mathcal{Z}'}$.

To each run of $\mathcal{F}$ with $\mathcal{S}$ and $\mathcal{Z}'$ we can correspond a run of $\mathcal{F}'$ with $\mathcal{S}'$ and $\mathcal{Z}'$; observe that the correspondence will preserve the history of the canonical functionality, i.e., the history of $\mathcal{F}$ in the run with $\mathcal{S}$ and $\mathcal{Z}'$ will be the same in the corresponding run of $\mathcal{F}'$ with $\mathcal{S}'$ and $\mathcal{Z}'$ (the environment is the same in both cases and $\mathcal{S}'$ operates identically to $\mathcal{S}$ in terms of the way it influences the functionality). Thus, given that the event that $\mathsf{validate}_2(\mathsf{history}) = 0$ happens with negligible probability over all runs of $\mathcal{F}$ with $\mathcal{S}$ and $\mathcal{Z}'$ (since this a real world simulation and whenever this event happens the functionality $\mathcal{F}$ returns an error symbol), it follows that it also happens with negligible probability over the runs of $\mathcal{F}'$ with $\mathcal{S}'$ and $\mathcal{Z}'$. Consider the event that $\mathcal{Z}'$ returns 1 over all runs of $\mathcal{F}$ with $\mathcal{S}$ and $\mathcal{Z}'$ and observe that its probability is the same to the event that $\mathcal{Z}'$ returns 1 over all runs of $\mathcal{F}'$ with $\mathcal{S}'$ and $\mathcal{Z}'$ where both events are taken over the conditional space where $\mathsf{validate}_2(\mathsf{history}) = 1$. Given that $\mathsf{validate}_2(\mathsf{history}) = 0$ is a negligible probability event in either space the proof of the theorem follows. □

Proof skeleton of Theorem 4.1

Proof. By contradiction, assume scheme Σ does not satisfy the property defined by game G. This means there exists an attacker A winning the game. To finish the proof, we need to present an environment $\mathcal{Z}$ which can distinguish the real from the ideal world with non-negligible probability. Based on the successful attacker A, we use $\mathcal{Z} = \mathcal{Z}_G^{\mathsf{A}}$ as defined in step 1 in Section 4.2.1. Note that in the real world, A is a successful attacker against the schemeΣ, so $\mathcal{Z}$ outputs 1 with non-negligible probability. However in the ideal world, the winning case would

cause any canonical functionality $\mathcal{F} \gtrsim \mathcal{F}_G$ to halt, so the environment $\mathcal{Z}$ can never output 1. Therefore the constructed $\mathcal{Z}$ distinguishes the two worlds with non-negligible probability. This finishes the proof. □

Proof of Lemma 5.2

Proof. (1) Denote by Y the language in the right hand side of the lemma's statement (1). First we need to show $B_{\mathtt{SIG},\mathrm{uf}} \subseteq Y$. Let w be any string in $B_{\mathtt{SIG},\mathrm{uf}}$; then it holds that there exist A, Σ such that w equals the history string in the ideal world execution of the environment $\mathcal{Z}_{\mathrm{uf}}^{\mathsf{A}}$ with adversary $\mathcal{S}_{\mathrm{uf}}^{\Sigma}$ and the dummy functionality $\mathcal{F}_{\mathtt{SIG}}^{\mathrm{dum}}$. Based on the definition of the environment $\mathcal{Z}_{\mathrm{uf}}^{\mathsf{A}}$ and the adversary $\mathcal{S}_{\mathrm{uf}}^{\Sigma}$, we know that the symbols $(\textsc{KeyGen}, \langle S, sid\rangle)(\textsc{KeyGenReturn}, \langle S, sid\rangle, vk)(\textsc{Sign}, \langle S, sid\rangle, m_1)(\textsc{SignReturn}, \langle S, sid\rangle, \sigma_1) \cdots (\textsc{Sign}, \langle S, sid\rangle, m_k)(\textsc{SignReturn}, \langle S, sid\rangle, \sigma_\ell)(\textsc{Verify}, \langle V, sid\rangle, \langle m', \sigma', vk\rangle)(\textsc{VerifyReturn}, \langle V, sid\rangle, 1)$ will be recorded into history in the dummy functionality. It follows that the string w belongs to the set Y.
Second we need to show $B_{\mathtt{SIG},\mathrm{uf}} \supseteq Y$. Let w be any string in Y. We will construct A, Σ such that in the ideal world execution of $\mathcal{Z}_{\mathrm{uf}}^{\mathsf{A}}$ with adversary $\mathcal{S}_{\mathrm{uf}}^{\Sigma}$ and the dummy functionality $\mathcal{F}_{\mathtt{SIG}}^{\mathrm{dum}}$ it holds that history $= w$. Given that $w \in Y$, there exist string $w = (\textsc{KeyGen}, \langle S, sid\rangle)(\textsc{KeyGenReturn}, \langle S, sid\rangle, vk)(\textsc{Sign}, \langle S, sid\rangle, m_1)(\textsc{SignReturn}, \langle S, sid\rangle, \sigma_1) \cdots (\textsc{Sign}, \langle S, sid\rangle, m_k)(\textsc{SignReturn}, \langle S, sid\rangle, \sigma_\ell)(\textsc{Verify}, \langle V, sid\rangle, \langle m', \sigma', vk\rangle)(\textsc{VerifyReturn}, \langle V, sid\rangle, 1)$. Define `gen` output $\langle vk, sk\rangle$. Define `sign` that upon input m_i returns σ_i for $1 \leq i \leq \ell$; Define A output $\langle m', \sigma'\rangle$; Define `verify` that upon input $\langle m', \sigma', vk\rangle$ returns 1. It follows immediately that the history string that in the ideal world execution of $\mathcal{Z}_{\mathrm{uf}}^{\mathsf{A}}$ with adversary $\mathcal{S}_{\mathrm{uf}}^{\Sigma}$ and the dummy functionality $\mathcal{F}_{\mathtt{SIG}}^{\mathrm{dum}}$ would equal w.

(2) It is easy to show the language $B_{\mathtt{SIG},\mathrm{uf}}$ is decidable. □

Proof of Theorem 5.5

Proof. Assume $\pi_{\Sigma(\mathtt{SIG})}$ cannot realize $\mathcal{F}_{\mathrm{uf}}$. This means there is a real world adversary $\mathcal{A}$ so that for all ideal world simulator $\mathcal{S}$ there exists an environment $\mathcal{Z}$ that can distinguish its interaction with $\mathcal{S}$ and $\mathcal{F}_{\mathrm{uf}}$ in the ideal world from that with $\mathcal{A}$ and $\pi_{\Sigma(\mathtt{SIG})}$ in the real world with non-negligible probability. Next we show how to construct A which can win the unforgeability game with non-negligible probability.

Since $\mathcal{Z}$ succeeds for any $\mathcal{S}$, it also succeeds for a generic $\mathcal{S}$ as described as follows. $\mathcal{S}$ runs a simulated copy of $\mathcal{A}$ internally, and forwards the communications between $\mathcal{A}$ and $\mathcal{Z}$; further $\mathcal{S}$ simulates protocol parties internally; whenever $\mathcal{S}$ receives a $(\textsc{LeakKeyGen}, \langle S, sid\rangle)$ from $\mathcal{F}_{\mathrm{uf}}$, it simulates a signer S to generate a key pair (vk, sk) by running the `gen`() algorithm, and return $(\textsc{InflKeyGen}, \langle S, sid\rangle, vk)$ command to the functionality; later whenever $\mathcal{S}$ receives a $(\textsc{LeakSign}, \langle S, sid\rangle, m)$ command from $\mathcal{F}_{\mathrm{uf}}$, it simulates the signer S to generate a signature σ for message m by running the `sign`() algorithm with the key sk, and return $(\textsc{InflSign}, \langle S, sid\rangle, \sigma)$ command to the functionality; whenever $\mathcal{S}$ receives a $(\textsc{LeakVerify}, \langle V, sid\rangle, \langle vk, m, \sigma\rangle)$ command from $\mathcal{F}_{\mathrm{uf}}$, it simulates a verifier V to generate the verification value ϕ by running the `verify`() algorithm over the tuple (vk, m, σ), and return $(\textsc{LeakVerify}, \langle V, sid\rangle, \phi)$ to the functionality; if at some point, $\mathcal{A}$ corrupts a party P, where P can be the signer or any verifiers, then $\mathcal{S}$ sends the corresponding $(\textsc{Corrupt}, P)$ command to $\mathcal{F}_{\mathrm{uf}}$.

Now we give the construction of A in detail. A simulates a copy of $\mathcal{Z}$ internally. It further tries to simulate for $\mathcal{Z}$ an interaction with the generic $\mathcal{S}$ and $\mathcal{F}_{\mathrm{uf}}$. As in $\mathcal{S}$, A runs a simulated copy of $\mathcal{A}$. But A does not simulate the signer to run the `gen`() and `sign`() algorithms to obtain the key pair, and further to generate signatures for messages as in $\mathcal{S}$; instead, whenever the $(\textsc{KeyGen}, \langle S, sid\rangle)$ command is activated by $\mathcal{Z}$, A uses the vk obtained from challenger C to form the $(\textsc{KeyGenReturn}, \langle S, sid\rangle, vk)$ command for $\mathcal{Z}$, and whenever an $(\textsc{Sign}, \langle S, sid\rangle, m)$ command is activated by $\mathcal{Z}$, A obtains the signature σ by interacting the challenger C with message m, and returns $(\textsc{SignReturn}, \langle S, sid, \rangle, \sigma)$ command to $\mathcal{Z}$; note that the challenger C queries the `sign`() oracle for signatures. For signature verification, A takes the similar way as the generic $\mathcal{S}$ did, i.e., whenever the command $(\textsc{Verify}, \langle V, sid\rangle, \langle vk, m, \sigma\rangle)$ is activated by $\mathcal{Z}$, A simulates a verifier V to generate the verification value $\phi \leftarrow \mathtt{verify}(vk, m, \sigma)$, and returns $(\textsc{VerifyReturn}, \langle V, sid, \rangle, \phi)$ command to $\mathcal{Z}$. At some point, if A finds a tuple $\langle vk, m, \sigma\rangle$ satisfies the following conditions: signer is not corrupted by $\mathcal{A}$, verification key vk is from the challenger, and m is not appeared in any queries with the challenger, and the verification value $\phi = 1$, then A

outputs (vk, m, σ) to the challenger C. Note that later C queries the $\texttt{verify}()$ oracle and obtains the verification value 1, and the judge J will decide that A wins the game.

We let F denote the event that in a run of $\pi_{\Sigma(\texttt{SIG})}$ with adversary $\mathcal{A}$ and environment $\mathcal{Z}$, a sequence w is generated such that $w \in B^{\text{ext}}_{\texttt{SIG},\text{uf}}$. Observe that if event F does not occur, the simulated $\mathcal{Z}$ cannot distinguish the two worlds. However, based on assumption above that $\mathcal{Z}$ can distinguish the two worlds with non-negligible probability, we know that event F must occur with non-negligible probability. Based on the construction of A, once F occurs, i.e., $w \in B^{\text{ext}}_{\texttt{SIG},\text{uf}}$, then A can obtain a subsequence $w' \in B_{\texttt{SIG},\text{uf}}$ which enables him to win the unforgeability game, i.e., A is a successful forger. □

Proof skeleton of Theorem 4.3

Proof. By contradiction, assume Σ does not satisfy the hiding property defined by G, i.e., there exists a successful attacker A who can guess the hidden bit b with non-negligible probability higher than $1/2$. Now we need to construct an environment that distinguish the real from the ideal world with non-negligible probability. Based on the successful attacker A, we use $\mathcal{Z} = \mathcal{Z}_G^{\mathsf{A}}$ as defined. Notice that in the real world, the protocol transcripts will be based on the bit B, and given A is a successful attacker, $\mathcal{Z}$ will output 1 with probability bounded away from $1/2$ by a non-negligible fraction; on the other hand, in the ideal world for any canonical functionality $\mathcal{F} \gtrsim \mathcal{F}_G$, since any such functionality will suppress "sensitive" part of the input which stops b from the adversary $\mathcal{S}$; now no matter how the adversary $\mathcal{S}$ is designed (note that $\mathcal{S}$ has adversarial role in this proof), the simulated protocol transcripts will be independently of b, therefore even an unbounded A will not be able to influence the output based on b. It follows that $\mathcal{Z}$ will output 1 with probability $1/2$. It follows that $\mathcal{Z}$ distinguishes the two worlds with non-negligible probability. □

Proof of Lemma 5.7

Proof. (1) Denote by Y the language in the right hand side of the lemma's statement (1). First we need to show $B_{\texttt{SIG},\text{comp}} \subseteq Y$. Let w be any string in $B_{\texttt{SIG},\text{comp}}$; then it holds that there exist A, Σ such that w equals the history string in the ideal world execution of the environment $\mathcal{Z}^{\mathsf{A}}_{\text{comp}}$ with adversary $\mathcal{S}^{\Sigma}_{\text{comp}}$ and the dummy functionality $\mathcal{F}^{\text{dum}}_{\texttt{SIG}}$. Based on the definition of the environment $\mathcal{Z}^{\mathsf{A}}_{\text{comp}}$ and the adversary $\mathcal{S}^{\Sigma}_{\text{comp}}$, we know that the symbols $(\text{KeyGen}, \langle S, sid\rangle)(\text{KeyGenReturn}, \langle S, sid\rangle, vk)(\text{Sign}, \langle S, sid\rangle, m)(\text{SignReturn}, \langle S, sid\rangle, \sigma)(\text{Verify}, \langle V, sid\rangle, \langle m, \sigma, vk\rangle)(\text{VerifyReturn}, \langle V, sid\rangle, 0)$ will be recorded into history in the dummy functionality. It follows that the string w belongs to the set Y.

Second we need to show $B_{\texttt{SIG},\text{comp}} \supseteq Y$. Let w be any string in Y. We will construct A, Σ such that in the ideal world execution of $\mathcal{Z}^{\mathsf{A}}_{\text{comp}}$ with adversary $\mathcal{S}^{\Sigma}_{\text{comp}}$ and the dummy functionality $\mathcal{F}^{\text{dum}}_{\texttt{SIG}}$ it holds that history $= w$. Given that $w \in Y$, there exists $w = (\text{KeyGen}, \langle S, sid\rangle)(\text{KeyGenReturn}, \langle S, sid\rangle, vk)(\text{Sign}, \langle S, sid\rangle, m)(\text{SignReturn}, \langle S, sid\rangle, \sigma)(\text{Verify}, \langle V, sid\rangle, \langle m, \sigma, vk\rangle)(\text{VerifyReturn}, \langle V, sid\rangle, 0)$. Define gen output $\langle vk, sk\rangle$. Define sign that upon input m returns σ; Define A output $\langle m, \sigma\rangle$; Define verify that upon input $\langle m, \sigma\rangle$ returns 0. It follows immediately that the history string that in the ideal world execution of $\mathcal{Z}^{\mathsf{A}}_{\text{comp}}$ with adversary $\mathcal{S}^{\Sigma}_{\text{comp}}$ and the dummy functionality $\mathcal{F}^{\text{dum}}_{\texttt{SIG}}$ would equal w.

(2) It is easy to show the language $B_{\texttt{SIG},\text{comp}}$ is decidable. □

Proof of Theorem 5.10

Proof. The proof strategy is very similar to the one for Theorem 5.5 above. Assume $\pi_{\Sigma(\texttt{SIG})}$ cannot realize $\mathcal{F}_{\text{comp}}$, i.e., there exists $\mathcal{A}$ so that for all $\mathcal{S}$ there exists an environment $\mathcal{Z}$ that can distinguish the two worlds with non-negligible probability. We again consider the generic $\mathcal{S}$ as constructed in proof of Theorem 5.5 above, and our goal here is to construct A which can win the completeness game with non-negligible probability.

We construct A by simulating a copy of $\mathcal{Z}$ internally. A further tries to simulate for $\mathcal{Z}$ an interaction with $\mathcal{S}$ and $\mathcal{F}_{\text{uf}}$. As in $\mathcal{S}$, A runs a simulated copy of $\mathcal{A}$, and the protocol parties as in the real world. At some point, if A finds

a tuple $\langle vk, m, \sigma\rangle$ satisfies the following conditions: signer is not corrupted by $\mathcal{A}$, a key pair (vk, sk) is honestly generated, and (m, σ) is an honestly generated message-signature pair, however the verification value $\phi = 0$, then A outputs (vk, m, σ) to the challenger C. Note that later C queries the $\mathtt{verify}()$ oracle and obtains the verification value 0, and the judge J will decide that A wins the game.

We define F as the event that in a run of $\pi_{\Sigma(\mathtt{SIG})}$ with $\mathcal{A}$ and $\mathcal{Z}$, a sequence w is generated such that $w \in B^{\text{ext}}_{\mathtt{SIG},\text{comp}}$. Observe that if the event F does not occur, the simulated $\mathcal{Z}$ cannot distinguish the two worlds. However, based on the assumption above that $\mathcal{Z}$ can distinguish the two worlds with non-negligible probability, we know that the event F must occur with non-negligible probability. Based on the construction of A described above, from $w \in B^{\text{ext}}_{\mathtt{SIG},\text{comp}}$, the attacker can obtain a subsequence $w' \in B_{\mathtt{SIG},\text{comp}}$, i.e., A can wins the completeness game. This means A is a successful completeness attacker. □

Proof of Lemma 5.12

Proof. (1) Denote by Y the language in the right hand side of the lemma's statement (1). First we need to show $B_{\mathtt{SIG},\text{cons}} \subseteq Y$. Let w be any string in $B_{\mathtt{SIG},\text{cons}}$; then it holds that there exist A, Σ such that w equals the history string in the ideal world execution of the environment $\mathcal{Z}^{\mathsf{A}}_{\text{cons}}$ with adversary $\mathcal{S}^{\Sigma}_{\text{cons}}$ and the dummy functionality $\mathcal{F}^{\text{dum}}_{\mathtt{SIG}}$. Based on the definition of the environment $\mathcal{Z}^{\mathsf{A}}_{\text{cons}}$ and the adversary $\mathcal{S}^{\Sigma}_{\text{cons}}$, we know that the symbols $(\textsc{KeyGen}, \langle S, sid\rangle)(\textsc{KeyGenReturn}, \langle S, sid\rangle, vk)$ $(\textsc{Verify}, \langle V_1, sid\rangle, \langle m, \sigma, vk\rangle)(\textsc{VerifyReturn}, \langle V_1, sid\rangle, \phi_1)$ $(\textsc{Verify}, \langle V_2, sid\rangle, \langle m, \sigma, vk\rangle)(\textsc{VerifyReturn}, \langle V_2, sid\rangle, \phi_2)$ where $\phi_1 \neq \phi_2$ will be recorded into history in the dummy functionality. It follows that the string w belongs to the set Y.

Second we need to show $B_{\mathtt{SIG},\text{cons}} \supseteq Y$. Let w be any string in Y. We will construct A, Σ such that in the ideal world execution of $\mathcal{Z}^{\mathsf{A}}_{\text{cons}}$ with adversary $\mathcal{S}^{\Sigma}_{\text{cons}}$ and the dummy functionality $\mathcal{F}^{\text{dum}}_{\mathtt{SIG}}$ it holds that history $= w$. Given that $w \in Y$, there exist string $w = (\textsc{KeyGen}, \langle S, sid\rangle)(\textsc{KeyGenReturn}, \langle S, sid\rangle, vk)$ $(\textsc{Verify}, \langle V_1, sid\rangle, \langle m, \sigma, vk\rangle)$ $(\textsc{VerifyReturn}, \langle V_1, sid\rangle, \phi_1)$ $(\textsc{Verify}, \langle V_2, sid\rangle, \langle m, \sigma, vk\rangle)$ $(\textsc{VerifyReturn}, \langle V_2, sid\rangle, \phi_2)$ with $\phi_1 \neq \phi_2$. Define $\mathtt{gen}$ output $\langle vk, sk\rangle$. Define A output $\langle m, \sigma\rangle$; Define $\mathtt{verify}$ that upon input $\langle m, \sigma, vk\rangle$ returns ϕ_1 for the first time and ϕ_2 for the second time. It follows immediately that the history string that in the ideal world execution of $\mathcal{Z}^{\mathsf{A}}_{\text{cons}}$ with adversary $\mathcal{S}^{\Sigma}_{\text{cons}}$ and the dummy functionality $\mathcal{F}^{\text{dum}}_{\mathtt{SIG}}$ would equal w.

(2) It is easy to show the language $B_{\mathtt{SIG},\text{cons}}$ is decidable. □

Proof of Theorem 5.15

Proof. The proof strategy is similar to the one for Theorem 5.5 above. Assume $\pi_{\Sigma(\mathtt{SIG})}$ cannot realize $\mathcal{F}_{\text{cons}}$, i.e., there exists $\mathcal{A}$ so that for all $\mathcal{S}$ there exists an environment $\mathcal{Z}$ that can distinguish the two worlds with non-negligible probability. Recall the generic $\mathcal{S}$ constructed in proof of Theorem 5.5 above, and here our goal is to construct A which can win the consistency game with non-negligible probability.

We construct A by simulating a copy of $\mathcal{Z}$ internally. A further tries to simulate for $\mathcal{Z}$ an interaction with $\mathcal{S}$ and $\mathcal{F}_{\text{uf}}$. As in $\mathcal{S}$, A runs a simulated copy of $\mathcal{A}$, and the protocol parties as in the real world. In the verification, if A finds a tuple $\langle vk, m, \sigma\rangle$ was verified twice and two different verification values are obtained, then A outputs (vk, m, σ) to the challenger C. Note that later C queries the $\mathtt{verify}()$ oracle twice with the tuple and obtains two different verification values, and the judge J will decide that A wins the game.

We define F as the event that in a run of $\pi_{\Sigma(\mathtt{SIG})}$ with $\mathcal{A}$ and $\mathcal{Z}$, a sequence w is generated such that $w \in B^{\text{ext}}_{\mathtt{SIG},\text{cons}}$. Observe that if the event F does not occur, the simulated $\mathcal{Z}$ cannot distinguish the two worlds. However, based on the assumption above that $\mathcal{Z}$ can distinguish the two worlds with non-negligible probability, we know that the event F must occur with non-negligible probability. Based on the construction of A described above, from $w \in B^{\text{ext}}_{\mathtt{SIG},\text{cons}}$, the attacker can obtain a subsequence $w' \in B_{\mathtt{SIG},\text{cons}}$, i.e., A can wins the completeness game. This means A is a successful consistency attacker. □

Proof of Proposition 5.22

Proof. First some observations. Note that in the [CLOS02] setting, the adversary is allowed to know all the communication between the functionality and the dummy parties except for the secret information, and it is in charge of message delivery. (Note also that "... and (sid) to $\mathcal{S}$,..." is redundant because the simulator is allowed to learn the header of the message.) Further, the Corrupt item is not explicitly shown in their functionality.

Now, to show the equivalence, we consider the "dummy" protocol π_{dummy}, which just forwards the input/output communication between the functionality and the environment, and we show that π_{dummy} in the $\mathcal{F}_{\mathtt{OT}}^{\mathrm{CLOS}}$-hybrid world (resp., $\mathcal{F}_{\mathtt{OT}}$-hybrid world) realizes functionality $\mathcal{F}_{\mathtt{OT}}$ (resp., $\mathcal{F}_{\mathtt{OT}}^{\mathrm{CLOS}}$).

– We first show that π_{dummy} in the $\mathcal{F}_{\mathtt{OT}}^{\mathrm{CLOS}}$-hybrid world realizes functionality $\mathcal{F}_{\mathtt{OT}}$. We need to construct a simulator $\mathcal{S}$ such that no $\mathcal{Z}$ can tell with non-negligible probability whether it interacts with $\mathcal{A}$ and π_{dummy} in the $\mathcal{F}_{\mathtt{OT}}^{\mathrm{CLOS}}$-hybrid world or with $\mathcal{S}$ and $\mathcal{F}_{\mathtt{OT}}$. The simulator $\mathcal{S}$ invokes a copy of $\mathcal{A}$ internally, and simulates for $\mathcal{A}$ the interaction with $\mathcal{Z}$ and the protocol π_{dummy} in the $\mathcal{F}_{\mathtt{OT}}^{\mathrm{CLOS}}$-hybrid world.

  In the case that no party is corrupted, whenever $\mathcal{S}$ receives $(\textsc{LeakTransfer}, \langle\langle S, R, sid\rangle, S\rangle)$ symbol from the "outside" functionality $\mathcal{F}_{\mathtt{OT}}$ (which means the functionality has an input $(\textsc{Transfer}, \langle\langle S, R, sid\rangle, S\rangle, \langle x_0, x_1\rangle)$ from the dummy sender), $\mathcal{S}$ sends $(\mathtt{sender}, sid, \cdot)$ to the internally simulated $\mathcal{F}_{\mathtt{OT}}^{\mathrm{CLOS}}$; note that $\mathcal{A}$ is allowed to see $(\mathtt{sender}, sid)$ but not its contents. Whenever $\mathcal{S}$ receives $(\textsc{LeakTransfer}, \langle\langle S, R, sid\rangle, R\rangle)$ from the $\mathcal{F}_{\mathtt{OT}}$ (which means the functionality has an input $(\textsc{Transfer}, \langle\langle S, R, sid\rangle, R\rangle, i)$ from the dummy receiver), $\mathcal{S}$ sends $(\mathtt{receiver}, sid, \cdot)$ to the simulated $\mathcal{F}_{\mathtt{OT}}^{\mathrm{CLOS}}$; note again that $\mathcal{A}$ can read the header $(\mathtt{receiver}, sid)$ but not the contents of the message. Now $\mathcal{S}$ simulates the inside functionality to send $(sid, \cdot)$ to the internally simulated receiver; again, note that $\mathcal{A}$ can read (sid) but not the content. Whenever $\mathcal{A}$ delivers the command $(sid, \cdot)$, $\mathcal{S}$ sends the outside functionality the symbol $(\textsc{InflTransfer}, \langle\langle S, R, sid\rangle, R\rangle)$.

  Next we discuss the cases where corruptions occur. Whenever $\mathcal{A}$ corrupts a party by sending a corruption command $(\mathtt{Corrupt}, S)$, $\mathcal{S}$ sends $(\textsc{Corrupt}, S)$ to the outside functionality $\mathcal{F}_{\mathtt{OT}}$. As a result, the outside functionality will return $\mathsf{history}_S$ to $\mathcal{S}$ and also S will be removed from the binding array, which means that $\mathcal{S}$ will be allowed to revise some part in history; note that such a revision should not violate the correctness restrictions defined by the extended bad languages (otherwise the validate predicate will trigger an error symbol which immediately would cause the simulation to fail). $\mathcal{S}$ reads $\mathsf{history}_S$ and if $(\textsc{Transfer}, \langle\langle S, R, sid\rangle, S\rangle, \langle x_0, x_1\rangle)$ has been recorded, then it simulates the inside functionality to reveal (x_0, x_1) to $\mathcal{A}$. In the case that $\mathcal{A}$ further supplies a pair (x_0', x_1'), and no $(sid, \cdot)$ has been delivered to the receiver, $\mathcal{S}$ by using $(\textsc{Patch}, \mathsf{history})$, will revise $(\textsc{Transfer}, \langle\langle S, R, sid\rangle, S\rangle, \langle x_0, x_1\rangle)$ into $(\textsc{Transfer}, \langle\langle S, R, sid\rangle, S\rangle, \langle x_0', x_1'\rangle)$; we note that the symbol $(\textsc{Transfer}, \langle\langle S, R, sid\rangle, S\rangle, \langle x_0, x_1\rangle)$ is allowed to be revised because the corresponding binding is empty given that party S is corrupted. Further, we note that at the moment the $(sid, \cdot)$ is delivered to the internal receiver, $\mathcal{S}$ will send an $\textsc{InflTransfer}$ symbol to the outside functionality, and a $\textsc{TransferReturn}$ symbol will be returned to the environment as described above. Now, although $(\textsc{Transfer}, \langle\langle S, R, sid\rangle, S\rangle, \langle x_0, x_1\rangle)$ is not marked, for the sake of the simulation, $\mathcal{S}$ will not revise this symbol into $(\textsc{Transfer}, \langle\langle S, R, sid\rangle, S\rangle, \langle x_0', x_1'\rangle)$, as otherwise the correction restriction would be violated.

  Next we consider case when the receiver is corrupted. Whenever $\mathcal{A}$ sends a command $(\mathtt{Corrupt}, R)$ to the inside functionality, $\mathcal{S}$ sends $(\textsc{Corrupt}, R)$ to the outside functionality $\mathcal{F}_{\mathtt{OT}}$. Now the outside functionality returns $\mathsf{history}_R$ to $\mathcal{S}$ and R will be removed from the binding array, and accordingly, $\mathcal{S}$ will be allowed to revise some parts of history; based on $\mathsf{history}_R$, $\mathcal{S}$ reconstructs the receiver's input and output and simulates the inside functionality to reveal such input and output to $\mathcal{A}$.

  This completes the construction of the simulator. We note that the simulation is perfect.

– We now show the other direction, i.e., that π_{dummy} in the $\mathcal{F}_{\mathtt{OT}}$-hybrid world realizes $\mathcal{F}_{\mathtt{OT}}^{\mathrm{CLOS}}$. We again need to construct a simulator $\mathcal{S}$ such that no $\mathcal{Z}$ can distinguish the two worlds with non-negligible probability. The construction is very similar to the one above. The simulator $\mathcal{S}$ invokes a copy of $\mathcal{A}$ internally, and simulates for $\mathcal{A}$ the interaction with $\mathcal{Z}$ and π_{dummy} in the $\mathcal{F}_{\mathtt{OT}}$-hybrid world. $\mathcal{S}$ interacts with the outside functionality $\mathcal{F}_{\mathtt{OT}}^{\mathrm{CLOS}}$.

  In the case of no corruptions, whenever $\mathcal{Z}$ inputs $(\mathtt{sender}, sid, x_0, x_1)$ to the dummy sender, $\mathcal{S}$ delivers the input to the outside functionality and learns the header $(\mathtt{sender}, sid)$, and it simulates the inside functionality $\mathcal{F}_{\mathtt{OT}}$ to send $(\textsc{LeakTransfer}, \langle\langle S, R, sid\rangle, S\rangle)$ to $\mathcal{A}$. Whenever $\mathcal{Z}$ inputs $(\mathtt{receiver}, sid, i)$ to the dummy receiver, $\mathcal{S}$ delivers the input to the outside functionality and learns the header $(\mathtt{receiver}, sid)$.

Now the functionality returns (sid, x_i) for the receiver, and further it simulates the inside functionality $\mathcal{F}_{\mathtt{OT}}$ to send (LEAKTRANSFER, $\langle\langle S, R, sid\rangle, R\rangle$) to $\mathcal{A}$. If both are received, and $\mathcal{A}$ returns (INFLTRANSFER, $\langle\langle S, R, sid\rangle, R\rangle$) to the inside functionality, $\mathcal{S}$ delivers (sid, x_i), which is produced by the outside functionality, to the receiver.

Next we discuss the cases when corruptions occur. Whenever $\mathcal{A}$ corrupts a party by sending a corruption symbol (CORRUPT, S), $\mathcal{S}$ sends $(\mathtt{Corrupt}, S)$ to the outside functionality $\mathcal{F}_{\mathtt{OT}}^{\mathrm{CLOS}}$. Now the outside functionality will return (x_0, x_1) if there is an input $(\mathtt{sender}, sid, x_0, x_1)$. $\mathcal{S}$ can construct $\mathsf{history}_S$ based on (x_0, x_1) and return it to $\mathcal{A}$. Similarly, whenever $\mathcal{A}$ sends out (CORRUPT, R), $\mathcal{S}$ can issue a $(\mathtt{Corrupt}, R)$ command and learn the receiver's input and output, and based on them construct $\mathsf{history}_R$ for $\mathcal{A}$. In the case that the sender is corrupted and no output has been received by the receiver, $\mathsf{history}_S$ can be revised into (TRANSFER, $\langle\langle S, R, sid\rangle, S\rangle, \langle x_0', x_1'\rangle$); note that this would not violate the correctness restriction. Now $\mathcal{S}$ operates as follows: $\mathcal{S}$ holds the input $(\mathtt{receiver}, sid, i)$ and until receives the revised information (x_0', x_1') from $\mathcal{A}$; then $\mathcal{S}$ delivers $(\mathtt{receiver}, sid, i)$ to the outside functionality and obtains the response which will be (sid, x_i'), and $\mathcal{S}$ delivers it to the receiver.

This concludes the simulation, and the simulation is perfect.

□